CONTROL
ENGINEERING

Dimitrios Hristu-Varsakelis
William S. Levine
Editors

Handbook of Networked and Embedded Control Systems

Birkhäuser

Control Engineering

Handbook of Networked and Embedded Control Systems

Dimitrios Hristu-Varsakelis
William S. Levine
Editors

Birkhäuser
Boston • Basel • Berlin

Dimitrios Hristu-Varsakelis
Department of Applied Informatics
University of Macedonia
Thessaloniki, 54006
Greece

William S. Levine
Department of Electrical and
Computer Engineering
University of Maryland
College Park, MD 20742
USA

Library of Congress Cataloging-in-Publication Data
Handbook of networked and embedded control systems / Dimitrios Hristu-Varsakelis,
William S. Levine, editors.
p. cm. – (Control engineering)
Includes bibliographical references and index.
ISBN 0-8176-3239-5 (alk. paper)
1. Embedded computer systems. I. Hristu-Varsakelis, Dimitrios. II. Levine, W. S. III.
Control engineering (Birkhäuser)

TK7895.E42H29 2005
629.8'9–dc22 2005041046

ISBN-10 0-8176-3239-5 e-BSN 0-8176-4404-0 Printed on acid-free paper.
ISBN-13 978-0-8176-3239-7

Printed in the United States of America. (JLS/MP)

9 8 7 6 5 4 3 2 1 SPIN 10925324

www.birkhauser.com

Contents

Preface

This handbook was motivated in part by our experience (and that of others) in performing research and in teaching about networked and embedded control systems (NECS) as well as in implementing such systems. Although NECS—along with the technologies that enable them—have become ubiquitous, there are few, if any, sources where a student, researcher, or developer can gain a sufficiently broad view of the subject. Oftentimes, the needed information is scattered in articles, websites, and specification sheets. Such difficulties are perhaps to be expected, given the relative newness of the subject and the diversity of its constitutive disciplines. From control theory and communications, to computer science and electronics, the variety of approaches, tools, and language used by experts in each field often acts as a barrier to understanding how ideas fit within the broader context of networked and embedded control.

With the above in mind, we have gathered a collection of articles that provide at least an introduction to the important results, tools, software, and technologies that shape the area of NECS. Our goal was to present the most important knowledge about NECS in a book that would be useful to anyone who wants to learn about any aspect of the subject. We hope that we have succeeded and that every reader will find valuable information in the book.

We thank the authors of each of the chapters. They are all busy people and we are extremely grateful to them for their outstanding work. We also thank Tom Grasso, Editor, Computational Sciences and Engineering at Birkhäuser Boston, for all his help in developing the handbook, and Regina Gorenshteyn, Assistant Editor, for guiding the editorial and production aspects of the volume. Lastly, we thank Torrey Adams whose copyediting greatly improved the book.

We gratefully acknowledge the support of our wives, Maria K. Hristu and Shirley Johannesen Levine, and our families.

College Park, MD
April 2005

Dimitrios Hristu-Varsakelis
William S. Levine

Part I

Fundamentals

Fundamentals of Dynamical Systems

William S. Levine

Department of ECE, University of Maryland, College Park, MD, 20742, U.S.A.
wsl@eng.umd.edu

1 Introduction

For the purposes of control system design, analysis, test, and repair, the most important part of the very broad subject known as system theory is the theory of dynamical systems. It is difficult to give a precise and sufficiently general definition of a dynamical system for reasons that will become evident from the detailed discussion to follow. All systems that can be described by ordinary differential or difference equations with real coefficients (ODEs) are indubitably dynamical systems. A very important example of a dynamical system that cannot be described by a continuous-time ODE is a pure delay. Most of this chapter will deal with different ways to describe and analyze dynamical systems. We will precisely specify the subclass of such systems for which each description is valid.

The idea of a system involves an approximation to reality. Specifically, a system is a device that accepts an input signal and produces an output signal. It is assumed to do this regardless of the energy or power in the input signal and independent of any other system connected to it. Physical devices do not normally behave this way. The response of a real system, as opposed to that of its mathematical approximation, depends on both the input power and whatever load the output is expected to drive.

Fortunately, the engineers who design real systems generally design them to behave as closely to an abstract system as possible. For electronic devices this amounts to creating subsystems with high input impedance and low output impedance. Such devices require minimal power in their inputs and will deliver the needed power to a broad range of loads without changing their outputs. Where this is not the case it is usually possible to purchase buffer circuits which will drive the load without altering the signal out of the original device. Good examples of this are the circuits used to connect the transistor-transistor logic (TTL) output of a typical microprocessor to a servomotor.

This means that, in both theory and practice, systems can be interconnected without worrying about either the input or output power. It also means

that a system can be completely described by the relation between its inputs and outputs without regard to the ways in which it is interconnected.

2 Continuous-Time Systems

We will limit our attention in this section to systems that can be described with sufficient accuracy by ordinary differential equations (ODEs). There are two different ways to describe such systems, in state space form or as an ODE relating the input and the output. In general the state space form is

$$\dot{\underline{x}}(t) = \underline{f}(\underline{x}(t), \underline{u}(t)) \tag{1}$$

$$\underline{y}(t) = \underline{g}(\underline{x}(t), \underline{u}(t)), \tag{2}$$

where $\dot{\underline{x}}(t)$ denotes the first derivative of the state n-vector $\underline{x}(t)$, $\underline{u}(t)$ is the m-vector input signal, and $\underline{y}(t)$ is the p-vector output signal; n,m, and p are integers; $\underline{f}(\cdot,\cdot)$ is some nonlinear function, as is $\underline{g}(\cdot,\cdot)$. The state vector is a complete set of initial conditions for the first-order vector ODE (1). One could be more general and allow both f and g to depend explicitly on time, but we will mostly ignore time-varying systems because of space limitations. We omit precise conditions on f and g needed to insure that there exists a unique solution to (1) for the same reason.

The state space form for a linear time-invariant (LTI) multi-input multi-output (MIMO) system is easily written. It is

$$\dot{\underline{x}}(t) = \underline{A}\underline{x}(t) + \underline{B}\underline{u}(t) \tag{3}$$

$$\underline{y}(t) = \underline{C}\underline{x}(t) + \underline{D}\underline{u}(t), \tag{4}$$

where the vectors $\underline{x}$, $\underline{y}$, and $\underline{u}$ are column n-, p-, and m-vectors respectively and all the matrices $\underline{A}$, $\underline{B}$, $\underline{C}$, and $\underline{D}$ have the appropriate dimensions. The solution of this exists and is unique for any initial condition $\underline{x}(0) = \underline{x}_0$ and any input signal $\underline{u}(t)$, for all $0 \leq t < t_f$.

It is worthwhile to be more precise about the meaning of "signal."

Definition 1. *A scalar continuous-time signal, denoted by* $\{u(t)$*for all* t*,* $t_0 \leq t < t_f\}$ *is a measurable mapping from an interval of the real numbers into the real numbers.*

The requirement that the mapping be measurable is a mathematical technicality that insures, among some more technical properties, that a signal can be integrated. We will generally be more casual and denote a signal simply by $u(t)$. An n-vector-valued signal is just an n-vector of scalar signals. More importantly, we assume that signals over the same time interval can be multiplied by a real scalar and added. That is, if $u(t)$ and $v(t)$ are both signals

defined on the same interval $t_0 \leq t < t_f$ and α and β are real numbers, then $w(t) = \alpha u(t) + \beta v(t)$ is also a signal defined on $t_0 \leq t < t_f$. Note that this assumption is true in regard to real signals. Physical devices that will multiply a signal by a real number (amplifiers) and add them (summers) exist. Because of this, it is natural to think of a signal as an element (a vector) in a vector space of signals.

The second ODE description (the first is the state space), in terms of only $\underline{y}(t)$, $\underline{u}(t)$, and their derivatives, is difficult to write in a general form. Instead, we show the general LTI single-input single-output (SISO) special case

$$\sum_0^n a_i \frac{d^i}{dt^i} y(t) = \sum_0^n b_i \frac{d^i}{dt^i} u(t), \tag{5}$$

where $a_i, b_i \in \mathbb{R}$. Because (5) is unchanged by division by a nonzero real number there is no loss of generality in assuming that $a_0 = 1$. Note that it is impossible to have a real physical system for which the highest derivative on the right-hand side n is greater than the highest derivative on the left-hand side.

There are three common descriptions of systems that are only valid for LTI systems, although there is an extension of the Fourier theory to nonlinear systems through Volterra series [1]. We present the SISO versions for simplicity and clarity. One is based on the Laplace transform, although the full power of the theory is not really needed for systems describable by ODEs. There are several versions of the Laplace transform. We use the bilateral or two-sided Laplace transform, defined by

$$Y(s) \stackrel{\text{def}}{=} \int_{-\infty}^{+\infty} y(t) e^{-st} dt, \tag{6}$$

because it is somewhat more convenient and useful for system theory [2]. The unilateral or single-sided Laplace transform is more useful for solving equations such as (5), but we are more interested in system theory than in the explicit solution of ODEs.

We regard the ODE (5) as the fundamental object because for many systems a description of the input-output behavior in terms of an ODE can be derived from the physics. Starting with (5) you need only that the Laplace transform of $\dot{y}(t) = sY(s)$, where $Y(s)$ denotes the Laplace transform of $y(t)$ and s is a complex number. Then, taking Laplace transforms of both sides of (5) gives

$$\sum_0^n a_i s^i Y(s) = \sum_0^n b_i s^i U(s). \tag{7}$$

Dividing both sides of (7) by $U(s)$ and by $\sum_0^n a_i s^i$ one obtains

$$\frac{Y(s)}{U(s)} = \frac{\sum_0^n b_i s^i}{\sum_0^n a_i s^i} \stackrel{\text{def}}{=} H(s). \tag{8}$$

Notice that $H(s)$, the transfer function of the system, completely describes the relation between the input $U(s)$ and the output $Y(s)$ of the system. It should be obvious that it is easy to go back and forth between $H(s)$ and the ODE in (5) by simply changing s to $\frac{d}{dt}$ and vice versa.

In fact, the Laplace transform makes it possible to write transfer functions for LTI systems that cannot be precisely described by ODEs. The most important example in control engineering is the system that acts as a pure delay. The transfer function for that LTI system is

$$H(s) = e^{-sT}, \tag{9}$$

where T is the time duration of the delay. However, the pure delay can be approximated to sufficient accuracy by an ODE using the Padé approximation (see "Control Issues in Systems with Loop Delays" by Mirkin and Palmor in this handbook).

Another common description of LTI systems is based on the Fourier transform. The great advantage of the Fourier transform is that, for a large class of real systems, it can be measured directly from the physical system. No mathematics is needed. To prove that this is so, start with either the ODE (5) or the transfer function (8). Let the input $u(t) = \cos(\omega t)$ for all $-\infty < t < \infty$. Using either standard ODE techniques or Laplace transforms—the transient portion of the response is ignored—the solution is found to be

$$y(t) = |H(j\omega)|\cos(\omega t + \angle H(j\omega)), \tag{10}$$

where $|H(j\omega)|$ denotes the magnitude of the complex number $H(s = j\omega)$ and $\angle H(j\omega)$ denotes its phase angle. $H(j\omega)$ is known as the frequency response of the system.

In the laboratory the input is zero prior to some starting time at which the input $u(t) = \cos(\omega t)$ for $t_0 \leq t < t_f$ is applied. One then waits until the initial transients die away and then measures the magnitude and phase of the output cosinusoid. This is repeated for a collection of values of ω and the gaps in ω are interpolated. Note that the presence in the output signal of any distortion or frequency content other than the input frequency indicates that the system is not linear.

One more way to describe an LTI system is based on the system's impulse response. Persisting in our view that the ODE is fundamental, we develop the impulse response by first posing two questions. What is the inverse Fourier transform of $H(j\omega)$, where $H(j\omega)$ is the transfer function of some LTI system? Furthermore, what is the physical meaning of this inverse transform? Note that the identical questions could be asked about the inverse Laplace transform of $H(s)$. The answer to the first question is simply a definition:

$$h(t) \stackrel{\text{def}}{=} \mathcal{F}^{-1}(H(j\omega)) \stackrel{\text{def}}{=} \int_{-\infty}^{\infty} H(j\omega)e^{j\omega t}d\omega/2\pi. \tag{11}$$

The answer to the second question is much more interesting. Think of the question this way. What input $u(t)$ will produce $h(t)$ as defined in (11)? The

answer is an input that is the inverse Fourier transform of $U(j\omega) = 1$ for all ω, $-\infty < \omega < \infty$. To see this, just write

$$Y(j\omega) = H(j\omega) \times 1. \tag{12}$$

The required signal is known as the unit impulse or Dirac delta function and is denoted by $\delta(t)$. Its precise meaning and interpretation require considerable mathematics and imagination [2, 3] although this discussion shows it must be, in some sense, the inverse Fourier transform of 1. In any case, this is why $h(t)$ as defined in (11) is known as the impulse response. It is a complete representation of the LTI system. Knowing the impulse response and the input signal, the output is computed from what is called a convolution integral,

$$y(t) = \int_{-\infty}^{\infty} h(t-\tau)u(\tau)d\tau. \tag{13}$$

Notice that the integral has to be computed for each value of t, $-\infty < t < \infty$, making the calculation of $y(t)$ by this means somewhat tedious.

The generalization of the Laplace and Fourier transforms and the impulse response and convolution integral to LTI MIMO systems is easy. One simply applies them term by term to the inputs and outputs. The impulse response can also be used on LTI systems, such as the pure delay of duration T ($h(t) = \delta(t-T)$), that cannot be written as ODEs as well as time-varying linear systems. The state space description also applies to time-varying systems.

For LTI systems that can be described by an ODE of the form (5), the ODE, transfer function $H(s)$, frequency response $H(j\omega)$, and impulse response $h(t)$ descriptions are completely equivalent. Knowing any one, you can compute any of the others. Given the state space description (3), it is possible to compute any of the other descriptions. We illustrate by computing $\underline{H}(s)$. Taking Laplace transforms of both sides of (3),

$$s\underline{X}(s) = \underline{A}\underline{X}(s) + \underline{B}\underline{U}(s) \tag{14}$$

$$(s\underline{I} - \underline{A})\underline{X}(s) = \underline{B}\underline{U}(s) \tag{15}$$

$$\underline{X}(s) = (s\underline{I} - \underline{A})^{-1}\underline{B}\underline{U}(s) \tag{16}$$

$$\underline{H}(s) = \underline{C}(s\underline{I} - \underline{A})^{-1}\underline{B} + \underline{D}. \tag{17}$$

The opposite direction, computing an $\underline{A}$, $\underline{B}$, $\underline{C}$, and $\underline{D}$ such that (17) holds given $\underline{H}(s)$ or its equivalent, is slightly more complicated. Many choices of $\underline{A}$, $\underline{B}$, $\underline{C}$, and $\underline{D}$ will produce the same $\underline{H}(s)$. They need not have the same number of states. The state space description is completely equivalent to the other descriptions if and only if it is minimal. The concepts of controllability and observability are needed to give the precise meaning of minimal. This will be discussed at the end of Section 4.

3 Discrete-Time Systems

There are exact analogs for discrete-time systems to each of the descriptions of continuous-time systems. The standard notation ignores the actual time completely and regards a discrete-time system as a mapping from an input sequence $u[k]$, $k_0 \leq k \leq k_f$ to an output $y[k]$, $k_0 \leq k \leq k_f$, where k is an integer. The discrete-time state space description is then

$$\underline{x}[k+1] = \underline{f}(\underline{x}[k], \underline{u}[k]) \tag{18}$$

$$\underline{y}[k] = \underline{g}(\underline{x}[k], \underline{u}[k]), \tag{19}$$

where $\underline{x}[k]$ is the state n-vector, $\underline{u}[k]$ is the m-vector input signal, and $\underline{y}[k]$ is the p-vector output signal; n, m, and p are integers.

A precise definition of a discrete-time signal is the following.

Definition 2. *A scalar discrete-time signal, denoted by* $\{u[k]$ *for all integers* k *such that* $k_0 \leq k < k_f\}$ *is a mapping from a set of consecutive integers into the real numbers.*

As for continuous-time signals, an n-vector signal is just an n-vector of scalar signals. The same scalar multiplication and addition apply in discrete time as in continuous time so discrete-time signals can also be viewed as vectors in a vector space of signals.

The LTI MIMO version is obviously

$$\underline{x}[k+1] = \underline{A}\underline{x}[k] + \underline{B}\underline{u}[k] \tag{20}$$

$$\underline{y}[k] = \underline{C}\underline{x}[k] + \underline{D}\underline{u}[k]. \tag{21}$$

The discrete-time analog of the ODE description fortuitously is known as an ordinary difference equation (ODE) or in statistics as an autoregressive moving average (ARMA) model. It has the form, in the SISO case,

$$\sum_0^n a_i y(t-i) = \sum_0^n b_i u(t-i), \tag{22}$$

where $a_i, b_i \in \mathbb{R}$.

There is a close analog and relative of the Laplace transform that applies to discrete-time systems. It is known as the Z-transform. As with the Laplace transform, we choose to work with the two-sided version which is, by definition,

$$X(z) \stackrel{\text{def}}{=} \sum_{m=-\infty}^{+\infty} x[m] z^{-m} \tag{23}$$

with z a complex number and $x[m]$, $-\infty < m < \infty$.

Similarly, there is a discrete-time Fourier transform. It is defined by the pair of equations

$$X(e^{j\omega}) \stackrel{\text{def}}{=} \sum_{k=-\infty}^{+\infty} x[k]e^{-j\omega k} \tag{24}$$

$$x[k] = \int_{2\pi} X(e^{j\omega})e^{j\omega k}d\omega/2\pi. \tag{25}$$

Notice that $X(e^{j\omega})$ is periodic in ω of period 2π. It is not possible to measure the discrete-time Fourier transform. It is possible to compute it very efficiently. Suppose you have a discrete-time signal that has finite duration—obviously something we could have measured as the output of a physical system:

$$x[k] = \begin{cases} x_k & \text{if } 0 \leq k \leq k_f - 1; \\ 0 & \text{otherwise.} \end{cases} \tag{26}$$

It is then possible [2,3] to define a discrete Fourier transform of $x[k]$ consisting of exactly k_f real numbers which we denote by $X_f[m]$ (the subscript f for Fourier):

$$X_f[m] = \frac{1}{k_f} \sum_{k=0}^{k_f-1} x[k]e^{-jm(2\pi/k_f)k}. \tag{27}$$

Applying the transforms to the ODE produces

$$H(z) = \frac{\sum_{i=0}^{n} b_i z^{-i}}{\sum_{i=0}^{n} a_i z^{-i}} \tag{28}$$

$$H(e^{j\omega}) = \frac{\sum_{i=0}^{n} b_k e^{-jk\omega}}{\sum_{i-0}^{n} a_k e^{-jk\omega}}. \tag{29}$$

Lastly, the pulse response is the discrete-time analog of the impulse response of continuous-time systems. There are no real difficulties. The pulse response $h[k]$ is just the output of an LTI system when the input is the discrete-time unit pulse, defined as

$$\delta[k] \stackrel{\text{def}}{=} \begin{cases} 1 & k = 0; \\ 0 & \text{otherwise.} \end{cases}$$

The generalizations and equivalences of these different descriptions of discrete-time systems are exactly the same as those for continuous-time systems, as described at the end of Section 2.

4 Properties of Systems

Two of the most important properties of systems are causality and stability. Loosely speaking, a system is causal if its response is completely determined

by its past and present inputs. The present output of a causal system does not depend on its future inputs. A similarly loose description of stability would be that small changes in the input or the initial conditions produce small changes in the output. Making these precise is fairly easy for LTI systems.

Definition 3. *A continuous-time LTI system is said to be* causal *if its impulse response* $h(t) = 0$ *for all* $t < 0$. *A discrete-time LTI system is* causal *if its pulse response* $h[k] = 0$ *for all* $k < 0$.

A more abstract and general definition of causality [4] begins by defining a family of truncator systems, P_T, defined for all real T by their action on an arbitrary input signal as

$$P_T(u(t)) \stackrel{\text{def}}{=} \begin{cases} u(t) & \text{for } t \leq T \\ 0 & \text{for } t > T. \end{cases} \tag{30}$$

Definition 4. *A system, denoted by* $\mathcal{S}$, *is causal if and only if* $P_T\mathcal{S} = P_T\mathcal{S}P_T$ *for all* T.

It is useful to distinguish two different forms of stability, although they are equivalent for LTI systems. The definitions are given for continuous-time; simply replace t by k for the discrete-time versions.

Definition 5. *A system is said to be* stable *if, with* $\underline{u}(t) = 0$ *for all* $t \geq 0$, *given any* $\epsilon > 0$ *there exists a* $\delta > 0$ *such that* $\|\underline{x}(t)\| < \epsilon$ *whenever* $\|\underline{x}_0\| < \delta$, *where* $\|\underline{x}(t)\|$ *denotes any norm of* $\underline{x}(t)$, *e.g.,* $\sqrt{\sum_1^n {x_i}^2}$. *The system is* asymptotically stable *if it is stable and* $\|\underline{x}(t)\| \to 0$ *as* $t \to 0$.

Definition 6. *A system is said to be* BIBO stable *(BIBO stands for bounded-input bounded-output) if* $\|\underline{y}(t)\| \leq M < \infty$ *whenever* $\|\underline{u}(t)\| \leq B < \infty$, *for some real numbers* M *and* B.

Notice that Definition 5 requires a state vector and depends crucially upon it. There are many elaborations of these two relatively simple definitions of stability. Many of these can be found in a textbook by H.K. Khalil [5].

There are several simple ways to determine if an LTI system is stable. Given the impulse (pulse) response, the following theorem applies [4, 6, 7].

Theorem 1. *A SISO continuous-time LTI system is BIBO stable if and only if* $\int_{-\infty}^{+\infty} |h(t)|dt \leq M < \infty$ *for some* M.

Replace the integral by an infinite sum to obtain the SISO discrete-time result. Replace the absolute value by a norm to generalize to the MIMO case.

Given either the ODE or the state space description of a system, causality has to be imposed as an extra condition. Differential and difference equations can generally be solved in either direction. For example, the ODE (5) could be solved for $y(0)$ from knowledge of a complete set of "initial" conditions at

t_f and $u(t)$ for all $0 \leq t < t_f$. Note that the backwards solution may not be unique in the discrete-time case.

Given either $H(z)$ or $H(s)$ causality is related to stability in an interesting way. A deeper understanding of the theory of transforms is needed here. Consider the two-sided (Laplace)Z-transform of a signal $y[k]$ ($y(t)$) for all $-\infty < k,t< +\infty$. It should be apparent from (23) ((6)) that the infinite sum (integral) may not converge for some values of z (s). For example, let the pulse response of an LTI system be

$$h_c[k] = \begin{cases} 0.9^k, & k \geq 0 \\ 0, & \text{otherwise.} \end{cases}$$

Then, using (23)

$$H_c(z) = \sum_{k=0}^{+\infty} (0.9/z)^k.$$

Computing the sum (using the fact that $\sum_{k=0}^{\infty} a^k = 1/(1-a)$ if $|a| < 1$) gives

$$H_c(z) = \frac{z}{z - 0.9}, \qquad \text{provided } |z| > 0.9.$$

Now, let the pulse response be

$$H_{ac}[k] = \begin{cases} -0.9^k, & k < 0 \\ 0 & \text{otherwise.} \end{cases}$$

Then

$$H_{ac}(z) = \frac{z}{z - 0.9}, \qquad \text{provided } |z| < 0.9.$$

Notice that two different pulse responses have the identical Z-transform if one ignores the region of the complex plane in which the infinite sum converges.

The key idea, as illustrated by the example, is that the region of the complex plane in which the Z-transform of a causal LTI system converges is the entire region outside of some circle of finite radius. The corresponding result for the Laplace transform is the region to the right of some vertical line in the complex plane. The next obvious question is: How is the boundary of that region determined?

To answer this question, we first assume for simplicity that $H(z)$ has the form of (28). We multiply numerator and denominator by z^n so we can work directly with polynomials in z. The denominator polynomial of $H(z)$ is then

$$p(z) = \sum_{i=0}^{n} a_i z^{n-i}. \tag{31}$$

As an nth-order polynomial in the complex number z with real coefficients, $p(z)$ has exactly n roots, i.e., values of z for which $p(z) = 0$. Simply replace z by s to obtain the corresponding continuous-time result.

Definition 7. *The* poles *of a SISO LTI system are the roots of its denominator polynomial.*

Note that this definition applies equally well to discrete- and continuous-time systems. For systems described by a transfer function of the form (28) or (8), the impulse, or pulse, response can be computed by first performing a partial fraction expansion of $H(z)$ or $H(s)$. For simplicity, we present the result for the case where $b_0 = 0$ and all the roots of the denominator polynomial are different—i.e., there are no repeated roots. Under these conditions,

$$H(z) = \frac{\sum_{i=1}^{n} b_i z^{n-i}}{\sum_{i=0}^{n} a_i z^{n-i}} = \sum_{i=1}^{n} A_i/(z - p_i), \tag{32}$$

where p_i denotes the ith pole of the system and A_i is the corresponding residue. Note that both A_i and p_i could be complex. If they are, then because a_i and b_i are real, the system must also have a pole that is the complex conjugate of p_i, and the residue of this pole must be the complex conjugate of A_i. Taking the inverse Z-transform of (32) gives

$$h[k] = \mathcal{Z}^{-1}(H(z)) = \sum_{i=1}^{n} A_i/(z - p_i)^i = \begin{cases} \sum_{i=1}^{n} A_i (p_i)^{k-1} & k \geq 1 \\ 0 & \text{otherwise.} \end{cases} \tag{33}$$

Applying Theorem 1 to (33) is the basic step in proving the following theorem.

Theorem 2. *A discrete-time (continuous-time) LTI system is asymptotically and BIBO stable if and only if all its poles, p_i, satisfy $|p_i| < 1$ ($\mathrm{Re}(p_i) < 0$).*

Similarly, the region of convergence of the Z-transform of a causal discrete-time LTI system is the region outside a circle of radius equal to $|p_m|$, where p_m is the pole with the largest absolute value. For Laplace transforms, it is the region to the right of p_m, the pole with the largest $\mathrm{Re}(p_m)$.

The numerator polynomial of $H(z)$ or $H(s)$ usually also has roots.

Definition 8. *The* finite zeros *of a SISO LTI system are the roots of its numerator polynomial.*

The reason for the adjective "finite" is rooted in the appropriate generalization of the definitions of poles and zeros to MIMO LTI systems. It is obvious from the definitions we have given that $|H(z)| = \infty$ at a pole and that $|H(z)| = 0$ at a zero of the system. This can be used to give more inclusive definitions of pole and zero. The one for a zero is particularly important.

Definition 9. *A* zero *of a SISO LTI discrete-time system is a value of z such that $H(z) = 0$. Similarly, a zero of a continuous-time SISO LTI system is a value of s such that $H(s) = 0$.*

With this definition of a zero, a system with n poles and m finite zeros can be shown to have exactly $n - m$ zeros at ∞. The zeros of a system are particularly important in feedback control because the zeros are invariant under feedback. That is, feedback cannot move a zero. Cancelling a zero or a pole is possible, as will be shown in the following section. However, understanding the ramifications of pole/zero cancellation requires at least two more concepts, controllability and observability.

Definition 10. *A time-invariant system is* completely controllable *if, given any initial condition* $\underline{x}(0) = \underline{x}_0$ *and any final condition* $\underline{x}(T) = \underline{x}_f$*, there exists a bounded piecewise continuous control* $u(t)$*,* $0 \leq t < T$ *for some finite* T *that makes* $\underline{x}(T) = \underline{x}_f$*.*

Definition 11. *A time-invariant system is* observable *if, given both* $\underline{y}(t)$ *and* $\underline{u}(t)$ *for all* $0 \leq t < T$ *for some finite* T*, it is possible to uniquely determine* $\underline{x}(0)$*.*

In both definitions it is assumed that the system is known. In particular, for LTI systems, $\underline{A}$, $\underline{B}$, $\underline{C}$, and $\underline{D}$ are known. There are also simple tests for controllability and observability for LTI systems.

Theorem 3. *An LTI system is controllable if and only if the* $n \times nm$ *matrix* $\underline{\mathcal{C}} = [\underline{B}\ \underline{AB}\ \underline{A}^2\underline{B}\ ...\ \underline{A}^{n-1}\underline{B}]$ *has rank* n*.*

Theorem 4. *An LTI system is observable if and only if the* $pn \times n$ *matrix*

$$\underline{\mathcal{O}} = \begin{bmatrix} \underline{C} \\ \underline{CA} \\ \underline{CA}^2 \\ \vdots \\ \underline{CA}^{n-1} \end{bmatrix} \tag{34}$$

has rank n*.*

As usual, there are many elaborations of the concepts of controllability and observability, providing precise extensions of these ideas to time-varying systems and further clarifying their meaning. Good sources for these more advanced ideas are the books by Kailath and Rugh [6, 7].

As mentioned earlier, given $\underline{H}(s)$ or its equivalent, the problem of finding $\underline{A}$, $\underline{B}$, $\underline{C}$, and $\underline{D}$ such that

$$\underline{H}(s) = \underline{C}(s\underline{(I)} - \underline{A})^{-1}\underline{B} + \underline{D} \tag{35}$$

has some subtleties. It is known in control theory as the realization problem and is covered in great detail in Kailath [6]. The SISO case is considerably simpler than the MIMO case. For brevity, we denote a state space model by its four matrices, viz. $\{\underline{A}, \underline{b}, \underline{c}, d\}$.

Definition 12. *A realization of a SISO transfer function $H(s)$ is* minimal *if it has the smallest number of state variables among all realizations of $H(s)$.*

Theorem 5. *A realization, $\{\underline{A}, \underline{b}, \underline{c}, d\}$, of $H(s)$ is minimal if and only if $\{\underline{A}, \underline{b}\}$ is controllable and $\{\underline{c}, \underline{A}\}$ is observable.*

All minimal realizations are equivalent, in the following sense.

Theorem 6. *Any two minimal realizations are related by a unique $n \times n$ invertible matrix of real numbers (i.e., a similarity transformation).*

The idea behind this theorem is that two n-dimensional state vectors are related by a similarity transformation. Specifically, if $\underline{x}_1$ and $\underline{x}_2$ are two n-vectors, then there exists an invertible matrix $\underline{P}$ such that $\underline{x}_2 = \underline{P}\underline{x}_1$. Define $\underline{x}_2(t) \stackrel{\text{def}}{=} \underline{P}\underline{x}_1(t)$. Differentiating both sides and making the obvious substitution gives

$$\dot{\underline{x}}_2(t) = \underline{P}\dot{\underline{x}}_1(t) = \underline{P}(\underline{A}\underline{x}_1(t) + \underline{b}u(t)). \tag{36}$$

Because $\underline{P}$ is invertible we can rewrite this as

$$\dot{\underline{x}}_2(t) = \underline{PAP}^{-1}\underline{x}_2(t) + \underline{Pb}u(t). \tag{37}$$

Applying this to the output equation shows that the following two realizations are equivalent in the sense that they have the same state dimension and produce the same transfer function:

$$\{\underline{A}, \underline{b}, \underline{c}, d\} \leftrightarrow \{\underline{PAP}^{-1}, \underline{Pb}, \underline{cP}^{-1}, d\}. \tag{38}$$

As will be demonstrated in the following section, it is possible to combine an LTI system with a pole at, say p_0, in series with an LTI system with a zero at the same value, p_0. The resulting transfer function could, theoretically, be reduced by cancelling the pole/zero pair, i.e., dividing out the common factor. It is not a good idea to perform this cancellation. The following theorem explains the difficulty.

Theorem 7. *A controllable and observable state space realization of a SISO transfer function $H(s)$ exists if and only if $H(s)$ has no common poles and zeros, i.e., no possible pole/zero cancellations.*

Thus, a SISO LTI system that has a pole zero cancellation must have at least one internal pole, i.e., a pole that cannot be seen from the input/output behavior of the system. If one attempts to cancel an unstable pole with a zero, the resulting system will be unstable even though this instability may not be evident from the linear input-output behavior. Generally, the instability will be noticed because it will drive the system out of its linear region.

The idea of pole/zero cancellations is formalized in the following definition.

Definition 13. *A SISO LTI system is* irreducible *if there are no pole/zero cancellations in its transfer function.*

In the SISO case, any minimal realization of an irreducible LTI system is completely equivalent to any other description of the system. Furthermore, the poles of the system are exactly equal to the eigenvalues of the $\underline{A}$ from any minimal realization. This allows us to write the following theorem linking all of the properties we have described.

Theorem 8. *The following statements are equivalent for causal irreducible SISO LTI systems:*

- *The system is BIBO stable*
- *The system's minimal realizations are controllable, observable, and asymptotically stable*
- *If the system is discrete-time, all its poles are inside the unit circle (have real part* < 0 *if continuous time).*

The MIMO generalizations of all of these results, including the definition and interpretation of zeros, and the meaning of irreducibility are vastly more complicated. See Kailath [6] for the details. There is a remarkable generalization of the idea of the zeros of a transfer function to nonlinear systems. An introduction can be found in an article by Isidori and Byrnes [8].

5 Interconnecting Systems

We will describe six ways to interconnect LTI systems in this section. The first three are exactly the same for discrete-time and continuous-time systems. The last three involve the interconnection of continuous-time and discrete-time systems. First, we consider the series connection of two LTI systems as shown in Fig. 1. The result is the transfer function

$$H(s) = H_1(s)H_2(s). \tag{39}$$

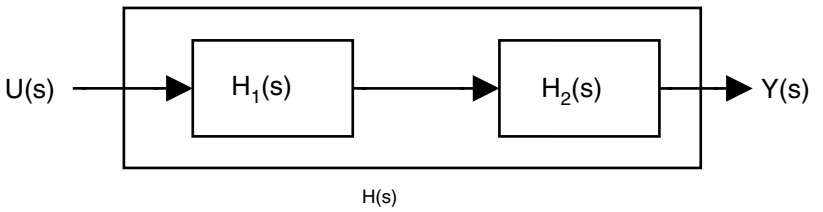


Fig. 1. The series interconnection of LTI systems

A proof is easy:

$$Y(s) = H_2(s)Y_1(s) = H_2(s)H_1(s)U(s). \tag{40}$$

As mentioned in the previous section, the series connection of an LTI system with a zero at p_0 with an LTI system with a pole at the same value p_0 results in their apparent cancellation in the transfer function, which is completely determined by the input-output behavior of the combined system. Cancellation of stable, well-behaved poles in this way is a common practice in control system design.

Two LTI systems connected in parallel are shown in Fig. 2. Notice that the figure introduces a new system, known variously as a summer, adder, or comparator. It is completely described by its operation. Its output $Y(s)$ is the sum of its two inputs $U_1(s) + U_2(s)$. Thus,

$$Y(s) = Y_1(s)+Y_2(s) = H_1(s)U(s)+H_2(s)U(s) = (H_1(s)+H_2(s))U(s). \tag{41}$$

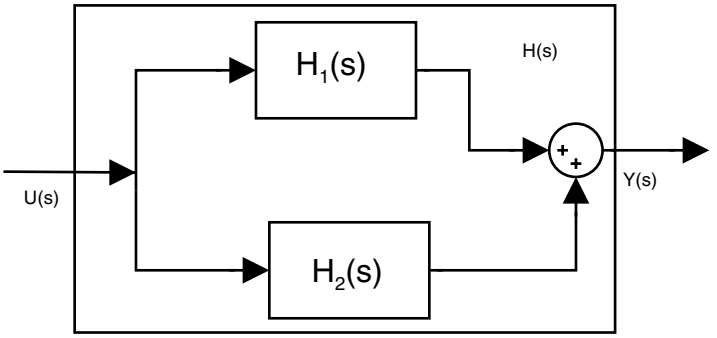


Fig. 2. The parallel interconnection of LTI systems

There is another way of combining subsystems, the feedback interconnection, illustrated in Fig. 3. Notice that the transfer function of the combined system is

$$H(s) = \frac{Y(s)}{U(s)} = \frac{H_1(s)}{1 + H_1(s)H_2(s)}. \tag{42}$$

This result can be derived by recognizing that $E(s) = U(s) - H_2(s)Y(s)$ and that $Y(s) = H_1(s)E(s)$, and doing some arithmetic.

Combining a discrete-time system in series with a continuous-time system requires an appropriate interface. If the output of the continuous-time system is input into the discrete-time system, then a sampler is needed. Conceptually this is simple. If $y(t)$ denotes the output of the continuous-time system and $u[k]$ denotes the input to the discrete-time system, then the sampler makes

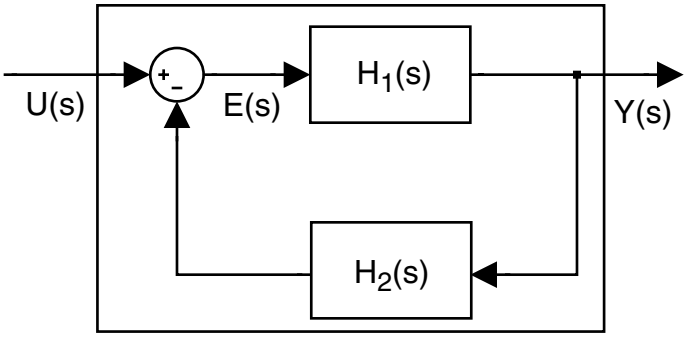


Fig. 3. The feedback interconnection of LTI systems

$$u[k] \stackrel{\text{def}}{=} y(kT_s), \tag{43}$$

where T_s is a fixed time interval known as the sampling interval and (43) holds for all integer k in some set of consecutive integers. Note that we are assuming the sampling interval is constant even though in many applications, especially involving embedded and networked computers, the sampling interval is not constant and can even fluctuate unpredictably. The theory is much simpler when T_s is constant. In fact T_s is often constant, and small fluctuations in the sampling interval can often be neglected. Note also that the series combination of a sampler and an LTI system is actually time varying.

One naturally expects sampling to lose information. Remarkably, it is theoretically possible to sample a continuous-time signal and still be able to reconstruct the original signal from its samples exactly, provided the sampling interval T_s is short enough. The precise details can be found in "Basics of Sampling and Quantization" by Santina and Stubberud in this handbook.

Combining a discrete-time system in series with a continuous-time system in the opposite order requires that the interface convert a discrete-time signal into a continuous-time one. Although there are several ways to do this, the most common and simplest way is to hold the discrete value for the whole sampling interval as shown below,

$$u(t) = y[k] \qquad \text{for all } t, \qquad kT_s \le t < (k+1)T_s. \tag{44}$$

The last of the six interconnections combines the previous two. It is the feedback interconnection of a discrete-time system with a continuous-time system. The problem is to characterize the combined system in a simple, precise, and convenient way. An exact discrete-time version of the continuous-time system can be obtained as follows. The solution to (3) starting from the initial condition $\underline{x}(t_0) = \underline{x}_0$ at $t = t_0$ is

$$\underline{x}(t) = e^{\underline{A}(t-t_0)}\underline{x}_0 + \int_{t_0}^{t} e^{\underline{A}(t-\tau)}\underline{B}\underline{u}(\tau)d\tau \tag{45}$$

$$\underline{y}(t) = \underline{C}\underline{x}(t) + \underline{D}\underline{u}(t), \tag{46}$$

where

$$e^{\underline{A}t} \stackrel{\text{def}}{=} \sum_{0}^{\infty} \frac{(\underline{A}t)^n}{n!}. \tag{47}$$

Applying these results when the initial condition is $\underline{x}(kT_s) = \underline{x}[k]$ and the input is $\underline{u}(t) = \underline{u}[k]$ for $kT_s \leq t < (k+1)T_s$ where T_s is the sampling interval gives

$$\underline{x}((k+1)T_s) = e^{\underline{A}T_s}\underline{x}[k] + \int_{k(T_s)}^{(k+1)T_s} e^{\underline{A}((k+1)T_s-\tau)}\underline{B}\underline{u}[k]d\tau. \tag{48}$$

Introducing the change of variables $\sigma = \tau - kT_s$ in the integral, replacing $\underline{x}((k+1)T_s)$ by $\underline{x}[k+1]$, and factoring out the constant $\underline{B}\underline{u}[k]$ gives

$$\underline{x}[k+1] = e^{\underline{A}T_s}\underline{x}[k] + \int_{0}^{(T_s} e^{\underline{A}(T_s-\sigma)}d\sigma\underline{B}\underline{u}[k]. \tag{49}$$

Define

$$\underline{A}_d \stackrel{\text{def}}{=} e^{\underline{A}T_s} \tag{50}$$

$$\underline{B}_d \stackrel{\text{def}}{=} \int_{0}^{T_s} e^{\underline{A}(T_s-\sigma)}d\sigma\underline{B} \tag{51}$$

$$\underline{C}_d \stackrel{\text{def}}{=} \underline{C} \tag{52}$$

$$\underline{D}_d \stackrel{\text{def}}{=} \underline{D}. \tag{53}$$

Then, we have the discrete-time system in state space form

$$\underline{x}[k+1] = \underline{A}_d\underline{x}[k] + \underline{B}_d u[k] \tag{54}$$

$$\underline{y}[k] = \underline{C}_d\underline{x}[k] + \underline{D}_d\underline{u}[k]. \tag{55}$$

Taking the Z-transform gives

$$\underline{H}(z) = \underline{C}_d(z\underline{I} - \underline{A}_d)^{-1}\underline{B}_d + \underline{D}_d. \tag{56}$$

Note that this basic approach will give a discrete-time system that is exactly equivalent to the sampled and held continuous-time system at the output sampling instants even if the sampling interval is not constant and is different at the input and the output. Systems of this type are often referred to as "sampled-data systems." See "Control of Single-Input Single-Output Systems" by Hristu and Levine in this handbook for another way to obtain an exact Z-transform for such a system.

There are many approximations to the exact result in (56) in the literature. This is partly for historical reasons. Many continuous-time control systems

were developed before cheap digital controllers became available. A quick and easy way to convert them to digital controllers was by means of a simple approximation to (56). Control and digital signal processing system designers also often use these approximations. The most commonly used and most useful of these is known variously as the trapezoidal method, Tustin's method, or the bilateral transformation. It is given by the following formula:

$$H(z) = H(s = (\frac{2}{T_s}\frac{z-1}{z+1})), \tag{57}$$

where T_s is the sampling interval, $H(s)$ is the continuous-time transfer function, and $H(z)$ is its discrete-time equivalent. More details can be found in many places, two of which are the chapter by Santina, Stubberud, and Hostetter [9] and a book by Franklin, Powell, and Workman [10].

6 Conclusion

This chapter is a very brief introduction to a very large subject. To learn more, it would be reasonable to begin with [2,3], which are undergraduate textbooks. The books by Kailath [6], Rugh [7], and Antsaklis and Michel [4] are graduate textbooks on linear systems. The book by Khalil [5] is a graduate text book on nonlinear systems. *The Control Handbook* [11] contains approximately 80 articles, each of which is a good starting point for learning about some aspect of dynamical systems and their control.

References

1. F. Lamnabhi-Lagarrique. Volterra and Fliess series expansions for nonlinear systems, in *The Control Handbook*, pp. 879–888, CRC Press, Boca Raton, FL, 1995.
2. A. V. Oppenheim and A. S. Willsky with S. Hamid Nawab. *Signals and Systems*, Prentice-Hall, Upper Saddle River, NJ, 2nd edition, 1997.
3. B. P. Lathi. *Linear Systems and Signals*, Oxford Unversity Press, New York, 2nd edition, 2005.
4. P. J. Antsaklis and A. N. Michel. *Linear Systems*, The McGraw-Hill Companies, New York, 1997.
5. H. K. Khalil. *Nonlinear Systems*, Prentice-Hall, Upper Saddle River, NJ, 3rd edition, 2002.
6. T. Kailath. *Linear Systems*, Prentice-Hall, Upper Saddle River, NJ, 1980.
7. W. J. Rugh. *Linear System Theory*, Prentice-Hall, Upper Saddle River, NJ, 2nd edition, 1995.
8. A. Isidori and C. J. Byrnes. Nonlinear zero dynamics, in *The Control Handbook*, pp. 917–923, CRC Press, Boca Raton, FL, 1995.
9. M. S. Santina, A. R. Stubberud, and G. H. Hostetter. Discrete-time equivalents to continuous-time systems, in *The Control Handbook* pp. 265–279, CRC Press, Boca Raton, FL, 1995.

10. G. F. Franklin, J. D. Powell, and M. Workman. *Digital Control of Dynamic Systems*, Addison-Wesley, San Diego, CA, 3rd edition, 1997.
11. W. S. Levine (Editor). *The Control Handbook*, CRC Press, Boca Raton, FL, 1995.

Control of Single-Input Single-Output Systems*

Dimitrios Hristu-Varsakelis[1] and William S. Levine[2]

[1] Department of Applied Informatics,
University of Macedonia, Thessaloniki, 54006, Greece dcv@uom.gr
[2] Electrical and Computer Engineering,
University of Maryland, College Park, MD 20742, U.S.A. wsl@umd.edu

1 Introduction

There is an extensive body of theory and practice devoted to the design of feedback controls for linear time-invariant systems. This chapter contains a brief introduction to the subject with emphasis on the design of digital controllers for continuous-time systems. Before we begin it is important to appreciate the limitations of linearity and of feedback. There are situations where it is best not to use feedback in the control of a system. Typically, this is true for systems that do not undergo much perturbation and for which sensors are either unavailable or too inaccurate. There are also limits to what feedback can accomplish. One of the most important examples is the nonlinearity that is present in virtually all systems due to the saturation of the actuator. Saturation will limit the range of useful feedback gains even when instability does not. It is important to keep this in mind when designing controllers for real systems, which are only linear within a limited range of input amplitudes.

The method used to design a controller depends critically on the information available to the designer. We will describe three distinct situations:

1. The system to be controlled is available for experiment but the designer cannot obtain a mathematical model of the system.
2. The designer has an experimentally determined frequency response of the system but does not have other modeling information.
3. The designer has a mathematical model of the system to be controlled.

The second case arises when the underlying physics of the system is poorly understood or when a reasonable mathematical model would be much too complicated to be useful. For example, a typical feedback amplifier might contain 20 or more energy storage elements. A mathematical model for this amplifier would be at least 20th order.

*This work was supported in part by NSF Grant EIA-008001.

It will be easiest to understand the different design methods if we begin with the third case, where there is an accurate mathematical model of the plant (the system to be controlled). When such a model is available, the feedback control of a single-input single-output (SISO) system begins with the following picture. The plant shown in Fig. 1 typically operates in continuous

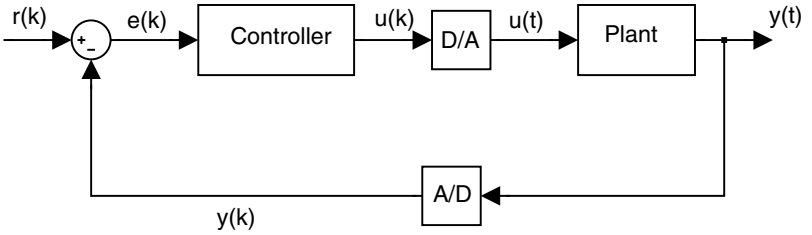


Fig. 1. A sampled-data feedback control system

time. It can be described by its transfer function:

$$Y(s) = G_c(s)U(s),$$

where $U(s)$, $Y(s)$ are the Laplace transforms of the input and output signals respectively, and

$$G_c(s) = \frac{\sum_0^{n-1} b_i s^i}{\sum_0^n a_i s^i} = \frac{\prod_1^{n-1}(s - z_i)}{\prod_1^n (s - p_i)}. \tag{1}$$

The coefficients a_i, b_i are real; the roots z_i, p_i of the numerator and denominator are the zeros and poles (respectively) of the transfer function. We assume that these parameters are given and that they are constant.

Note that (1) limits the class of systems to those that can be adequately approximated by such a transfer function. For a discussion of controller design when $G_c(s)$ includes a pure delay, described by e^{-sT}, see "Control Issues in Systems with Loop Delays" by Mirkin and Palmor in this handbook. The output in Fig. 1 is fed directly back to the summer (comparator). For simplicity and clarity we restrict our discussion to unity feedback systems, as in Fig. 1. It is fairly easy to account for dynamics or filtering associated with the sensor if necessary.

The controller (in cascade with the plant) is to be designed so that the closed-loop system meets a given set of specifications. The controller is assumed to be linear (in a sense to be made precise shortly). Modern controllers are often implemented in a digital computer. This requires the use of analog-to-digital (A/D) and digital-to-analog (D/A) converters in order to interface with the continuous-time plant. This makes the plant, as seen by the

controller, a sampled-data system with input $u(k)$ and output $y(k)$. See the chapter, in this handbook, entitled "Basics of Sampling and Quantization" by Santina and Stubberud for a discussion of the effects of time discretization and D/A and A/D conversion.

2 Description of Sampled-Data Systems

The D/A block shown in Fig. 1 converts the discrete-time signal $u(k)$ produced by the controller to a continuous-time piecewise constant signal via a "zero-order hold" (ZOH). Let $u(k)$ be the discrete-time input signal, arriving at the D/A block at multiples of the sampling period T. In the time domain, the ZOH can be modeled as a sum of shifted unit step functions:[3]

$$u(t) = \sum_{0}^{\infty} u(k)[1(t - kT) - 1(t - (k + 1)T)].$$

The Laplace transform of the last expression yields

$$U(s) = \underbrace{\sum_{0}^{\infty} u(k)e^{-kTs}}_{U(z)} \left(\frac{1}{s} - \frac{e^{-Ts}}{s}\right).$$

If we think of $u(k)$ as a continuous-time impulse train, $u(k)\delta(t - kT)$, then the ZOH has a transfer function

$$G_{ZOH}(s) = \frac{1}{s}(1 - e^{-sT}).$$

From the point of view of the (discrete-time) controller, the transfer function of the sampled-data system is given by the z-transform of the ZOH/plant system

$$G(s) = \frac{1 - e^{-sT}}{s} G_c(s),$$

which is (by $z = e^{sT}$)

$$G(z) = \frac{z - 1}{z} \mathcal{Z}\{G_c(s)/s\},$$

where $\mathcal{Z}\{G_c(s)/s\}$ is computed by first calculating the inverse Laplace transform of $G_c(s)/s$ to obtain a continuous-time signal, $\hat{g}(t)$, then sampling this signal, and finally computing the Z-transform of this discrete-time signal.

If we let $C(z)$ denote the transfer function of the controller, then the closed-loop transfer function is

[3]The unit step function $1(t)$ equals zero for $t < 0$, one for $t \geq 0$.

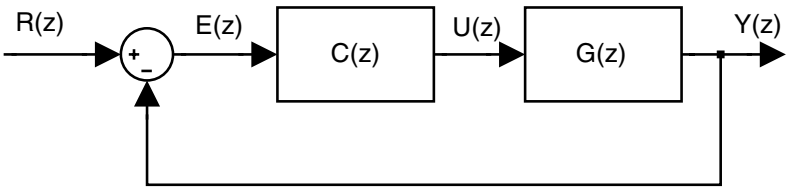


Fig. 2. A sampled-data feedback control system

$$\frac{Y(z)}{U(z)} = G_{cl}(z) = \frac{G(z)C(z)}{1 + G(z)C(z)}.$$

This is illustrated in Fig. 2.

An important point to remember about sampled-data systems is that the real system evolves in continuous time, including the time between the sampling instants. This inter-sample behavior must be accounted for in most applications.

3 Control Specifications

The desired performance of the closed-loop system in Fig. 2 is usually described by means of a collection of specifications. They can be organized into four groups:

- Stability
- Steady-state error
- Transient response
- Robustness

These will be discussed in order below.

3.1 Stability

A system is *bounded-input bounded-output (BIBO) stable* if any bounded input results in a bounded output. A system is *internally stable* if its state decays to zero when the input is identically zero. If we limit ourselves to linear time-invariant (LTI) systems, then all questions of stability can be settled easily by examining the poles of the closed-loop system. In particular, the closed-loop system is both BIBO and internally stable if and only if all of its poles[4] are inside the unit circle. Mathematically, if the poles of the closed-loop system are denoted by p_i, $i = 1, 2, ..., n$ then the system is BIBO and internally stable if $|p_i| < 1$ for all i.

[4] This must include any poles that are cancelled by zeros.

3.2 Steady-state error

In many situations the main objective of the closed-loop system is to track a desired input signal closely. For example, a paper-making or metal-rolling machine is expected to produce paper or metal of a specified thickness. Brief, transient errors when the process starts, while undesirable, can often be ignored. On the other hand, persistent tracking errors are a serious problem. Typically, the specification will be that the steady-state error in response to a unit step input must be exactly zero. It is surprisingly easy to meet this requirement in most cases.

The difference between input and output is $e(k)$, or in the z-domain,

$$E(z) = \frac{R(z)}{1 + G(z)C(z)}.$$

We can examine the steady-state error by using the "final value theorem"

$$e(\infty) \triangleq \lim_{k\to\infty} e(k) = \lim_{z\to 1}(1 - z^{-1})E(z).$$

If the input is a unit step ($U_s(z) = z/(z-1)$), then the last equation yields

$$e(\infty) = \lim_{z\to 1} \frac{1}{1 + G(z)C(z)}. \tag{2}$$

Equation (2) indicates that the steady-state error will be zero provided that

$$\lim_{z\to 1} G(z)C(z) = \infty,$$

which will be true if $G(z)C(z)$ has one or more poles at $z = 1$.

More elaborate steady-state specifications exist, but the details can easily be derived using this example as a model or by consulting the books by Dorf and Bishop [5] or Franklin et al. [6].

3.3 Transient response

The transient response of the closed-loop system is important in many applications. A good example is the stability and control augmentation systems (SCASs) now common in piloted aircraft and some automobiles. These are systems that form an inner (usually multi-input multi-output (MIMO)) control loop that improves the handling qualities of the vehicle. The pilot or driver is the key component in an outer control loop that provides command inputs to the SCAS. The transient characteristics of the vehicle are crucial to the pilot's and driver's handling of the vehicle and to the passenger's perception of the ride. If you doubt this, imagine riding in or driving a car with a large rise time or large percent overshoot (defined below).

The transient response of an LTI system depends on the input as well as on the initial conditions. The standard specifications assume a unit step as the test input, and the system starts from rest, with zero initial conditions. The resulting step response is then characterized by several of its properties, most notably its rise time, settling time, and percent overshoot. These are displayed in Fig. 3 and defined below.

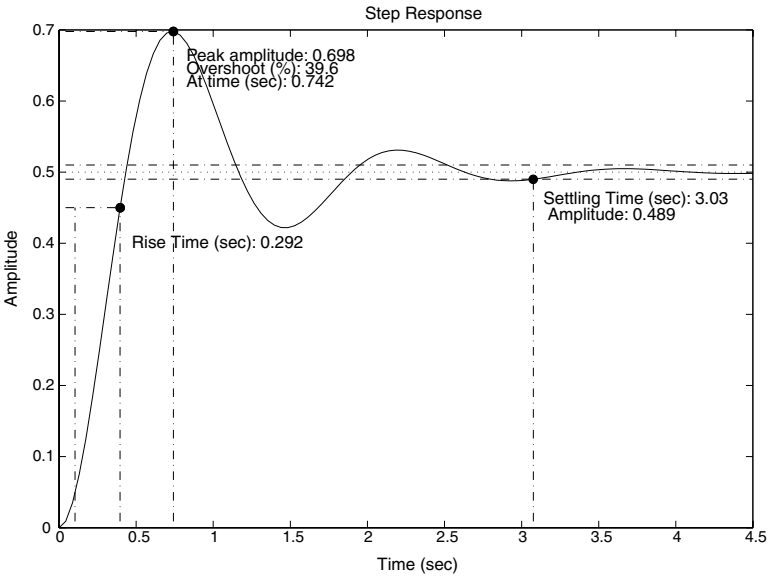


Fig. 3. The step response of an LTI system and its properties

- Rise time: Usually defined to be the time required for the step response to go from 10% of its final value to 90% of its final value.
- Settling time: Usually defined to be the time at which the step response last crosses the lines at $\pm 2\%$ of its final value.
- Percent overshoot: Usually defined to be the ratio (peak amplitude minus final value)/(final value) expressed as a percentage.

In each case there are variant definitions. For example, sometimes $\pm 1\%$ or $\pm 5\%$ is used instead of $\pm 2\%$ in the definition of settling time. The final value is the steady-state value of the step response, 0.5 in Fig. 3.

3.4 Robustness

Because a system either is or is not stable, a nominally stable system may become unstable as a result of arbitrarily small differences between the nominal plant $G(z)$ used for design and the real plant. Such differences might be

due to inaccuracies in parameter values, variations in operating conditions, or the deliberate omission of aspects of the nominal plant. For example, the flexure modes of the body and wings of an aircraft are usually omitted from the nominal plant model used for controller design. This underscores the importance of knowing how "close" to instability the closed-loop system is. The "distance to instability" is commonly quantified for SISO LTI systems in two ways. One is the *gain margin*, namely the gain factor K that must be applied to the forward path (replacing $G(z)C(z)$ by $KG(z)C(z)$ in Fig. 2) in order for the system to become unstable. The other, known as the *phase margin*, is the maximum amount of delay (or phase shift) $e^{-j\phi_M}$ that can be introduced in the forward path before the onset of instability.

Robustness, as a specification and property of a controlled system, has received much attention in the research literature in recent years. This has led to robustness tests for MIMO systems as well as a variety of tools for designing robust control systems. See [8, 15] for more details.

4 Analysis and Design Tools

4.1 The root locus

Consider making the controller in Fig. 2 simply a gain, i.e., $C(z) = K$.

As K varies from 0 to ∞, the poles of $G_{cl}(z) = \frac{KG(z)}{1+KG(z)}$ trace a set of curves (called the "root locus") in the complex plane. When $K = 0$ the poles of the "closed-loop system" are identical to the poles of the open-loop system, $G(z)$. Thus, each locus starts at one of the poles of $G(z)$. As $K \to \infty$ it is possible to prove that the closed-loop poles go to the open-loop zeros, including both the finite and infinite zeros, of $G(z)$. Given a specific value for K, it is easy to compute the resulting closed-loop pole locations. Today, one can easily compute the entire root locus; for example, the MATLAB command `rlocus` was used to produce Fig. 4. The root locus plot is obviously useful to the designer who plans on using a controller $C(z) = K$. He or she simply chooses a desirable set of pole locations, consistent with the loci, and determines the corresponding value of K. MATLAB has a command, `rlocfind`, that facilitates this. Alternatively, one can use the sisotool graphical user interface (GUI) in MATLAB to perform the same task. The choice of pole location is aided by the use of a grid that displays contours of constant natural frequency and damping ratio. We will have more to say regarding the choice of pole locations and the use of the root locus plot in Section 5.1.

By combining the controller and the plant and multiplying by K (the effective plant is then $C(z)G(z)$), the root locus can be used to determine the gain margin. As will be explained later, the effect of various compensators can also be analyzed and understood by appropriate use of the root locus. Lastly, the idea of the root locus, the graphical display of the pole locations as an implicit function of a single variable in the design, can be very useful in

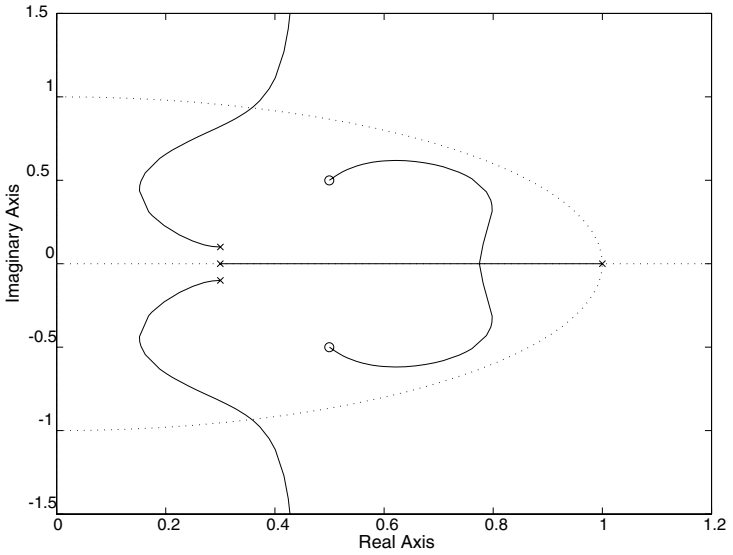


Fig. 4. The root locus when $G(z) = \frac{z^2 - z + 0.5}{z^4 - 1.9z^3 + 1.18z^2 - 0.31z + 0.03}$

a variety of applications. Modern computers make it fairly easy to generate such root loci.

4.2 The Bode, Nyquist, and Nichols plots

There are at least two situations where it is preferable to use the frequency response of the plant rather than its transfer function $G(z)$ for control system design. First, when the plant is either stable or easily stabilized, it is often possible to determine $|G(e^{j\Omega T})|$ and $\angle G(e^{j\Omega T})$, where T is the time interval between samples, experimentally for a range of values of Ω. This data is sufficient for control design, completely eliminating the need for an analytical expression for $G(z)$. Second, a system with many poles and zeros can produce a very complicated and confusing root locus. The frequency response plots of such a system can make it easier for the designer to focus on the essentials of the design. This second situation is exemplified by feedback amplifier design, where a state space or transfer function model would be of high order, but the frequency response is relatively simple.

The Nyquist plot of the imaginary part of $G(e^{j\Omega T})$ versus the real part of $G(e^{j\Omega T})$ provides a definitive test for stability of the closed-loop system. It also gives the exact gain and phase margins unambiguously. However, it is not particularly easy to use for design. In contrast, both the Bode plots and Nichols chart are very useful for design but can be ambiguous with regard to stability. There are two Bode plots. The Bode magnitude plot presents

$20\log|G(e^{j\Omega T})|$ on the vertical axis versus $\log\Omega$ on the horizontal axis. The Bode phase plot shows $\angle G(e^{j\Omega T})$ on the vertical axis and uses the same horizontal axis as the magnitude plot. The Nichols chart displays $20\log|G(e^{j\Omega T})|$ on the vertical axis versus $\angle G(e^{j\Omega T})$ on the horizontal axis. An example of both plots is shown in Fig. 5. Note that the lightly dotted curves on the Nichols chart are contours of constant gain (in decibels) and phase (in degrees) of the closed-loop system. Thus, any point on the Nichols plot for $G(z)$ also identifies a value of $20\log|\frac{G(z)}{(1+G(z))}|$ and of $\angle\frac{G(z)}{(1+G(z))}$ for some value of Ω.

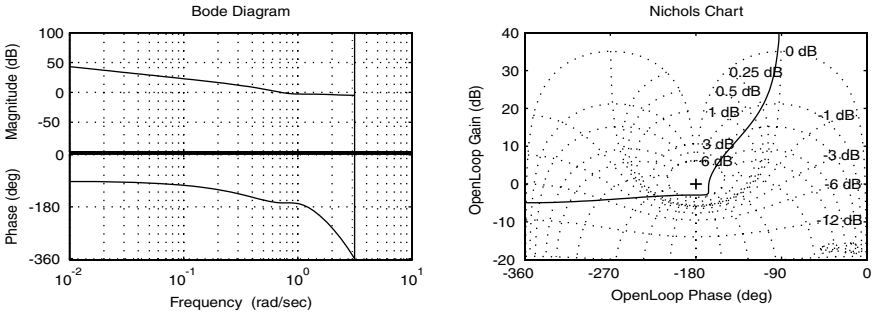


Fig. 5. The Bode plots and Nichols chart for $G(z) = \frac{z^2-z+0.5}{z^4-1.9z^3+1.18z^2-0.31z+0.03}$

The use of logarithmic scaling for the magnitude offers an important convenience: The effect of a series compensator $C(z)$ on the logarithmic magnitude is additive, as is its effect on the phase.

5 Classical Design of Control Systems

In reality, the design of a control system usually includes the choosing of sensors, actuators, computer hardware and software, A/D and D/A converters, buffers, and, possibly, other components of the system. In a modern digital controller the code implementing the controller must also be written. In addition, most control systems include a considerable amount of protection against emergencies, overloads, and other exceptional circumstances. Lastly, it is now common to include some collection and storage of maintenance information as well. Although control theory often provides useful guidance to the designer in all of the above-mentioned aspects of the design, it only provides explicit answers for the choice of $C(z)$ in Fig. 2. It is this aspect of control design that is covered here.

5.1 Analytical model-based design

The theory of control design often begins with an explicitly known plant $G(z)$ and a set of specifications for the closed-loop system. The designer is

expected to find a controller $C(z)$ such that the closed-loop system satisfies those specifications. In this case, a natural beginning is to plot the root locus for $G(z)$. If the root locus indicates that the specifications can be met by a controller $C(z) = K$, then the theoretical design is done. However, it is not a trivial matter to determine from the root locus if there is a value of K for which the specifications are met. Notice that the example specifications in Section 3 include both time domain and frequency domain requirements.

The designer typically needs to be able to visualize the closed-loop step response from knowledge of the closed-loop pole and zero locations only. This is easily done for second-order systems where there is a tight linkage between the pole locations and transient response. Many SISO controlled systems can be adequately approximated by a second-order system even though the actual system is of higher order. For example, there are many systems in which an electric motor controls an inertia. The mechanical time constants in such a system are often several orders of magnitude slower than the electrical ones and dominate the behavior. The electrical transients can be largely ignored in the controller design.

A second-order system can be put in a standard form that only depends on two parameters, the damping ratio ζ and the natural frequency ω_n. The continuous-time version is

$$G_{cl}(s) = \frac{{\omega_n}^2}{s^2 + 2\zeta\omega_n s + {\omega_n}^2}, \tag{3}$$

where $G_{cl}(s)$ denotes the closed-loop transfer function. Notice that the poles of $G_{cl}(s)$ are located at $s = -\zeta\omega_n \pm j\omega_n\sqrt{1-\zeta^2} = \omega_n e^{j\pi\pm\cos^{-1}\zeta}$. For stable systems with a pair of complex conjugate poles, $0 \leq \zeta < 1$. The description (3) is not used for systems with real poles. The system (3) has step response

$$y(t) = 1 - \frac{e^{-\zeta\omega_n t}}{\sqrt{1-\zeta^2}}\left(\sin(\sqrt{1-\zeta^2}\omega_n t + \tan^{-1}(\frac{\sqrt{1-\zeta^2}}{\zeta}))\right). \tag{4}$$

The constants ζ and ω_n completely determine the step response. With a little experience a designer can then picture the approximate step response in his or her mind while looking at the pole locations. For a system with additional poles and zeros the actual step response can be quite different from that in (4), but designers need insight and a way to start. An initial design that is very far from meeting the specifications can often be modified and adjusted into a good design after several iterations.

It is possible to create a second-order discrete-time system whose step response exactly matches that of (4). The first step is to choose a time interval between outputs of the discrete-time system, say T_s. Then, if the continuous-time system has a pole at p_i, the corresponding discrete-time system must have a corresponding pole at $p_{id} = e^{p_i T_s}$. The poles of the continuous-time system (3) are at $p_i = -\zeta\omega_n \pm j\omega_n\sqrt{1-\zeta^2}$. Thus, the poles of the discrete-time system are at $p_{id} = e^{-\zeta\omega_n T_s}e^{\pm j\omega_n\sqrt{1-\zeta^2}T_s}$. Writing the p_{id} in polar form

as $R \cdot e^{j\theta}$ (the subscripts have been dropped because there is only one value) gives

$$R = e^{-\zeta\omega_n T_s} \tag{5}$$

$$\theta = \pm\omega_n\sqrt{1-\zeta^2}T_s. \tag{6}$$

Solving explicitly for ζ and ω_n gives

$$\zeta = \pm\frac{\ln(R)}{\sqrt{\theta^2 + (\ln(R))^2}} \tag{7}$$

$$\omega_n = \pm\sqrt{\theta^2 + (\ln(R))^2}. \tag{8}$$

This defines two curves in the z-plane, a curve of constant ζ and a curve of constant ω_n. These curves can be plotted on the root locus plot—the MATLAB command is `zgrid`. For a second-order system in the standard form (3), both the transient response characteristics and the phase margin are directly related to ζ and ω_n:

$$rise\ time = t_r \approx \frac{1.8}{\omega_n} \tag{9}$$

$$settling\ time = t_s \approx \frac{4.6}{\zeta\omega_n} \tag{10}$$

$$percent\ overshoot = P.O. = 100\frac{e^{-\pi\zeta/\sqrt{1-\zeta^2}}}{final\ value}. \tag{11}$$

The $final\ value$ is the constant steady-state value reached after the transients have died out ($final\ value = \lim_{k\to\infty} y(k)$).

Clearly, if a designer can satisfy the specifications using only $C(z) = K$, the best value of K can be chosen by plotting the root locus of $G(z)$ and looking at where the loci intersect the contours of constant ζ and ω_n. If this is not sufficient, there are several standard components one can try to include in $C(z)$ in order to alter the root locus so that its branches pass through the desired values of ζ and ω_n. The best known of these are the lead and lag compensators defined here for discrete-time systems.

Lead compensator:

$$C_{le}(z) = \frac{(\frac{z}{z_l} - 1)}{(\frac{z}{p_l} - 1)}, \quad 0 \le p_l < z_l \le 1 \tag{12}$$

Lag compensator:

$$C_{la}(z) = \frac{(\frac{z}{z_l} - 1)}{(\frac{z}{p_l} - 1)}, \quad 0 \le z_l < p_l \le 1. \tag{13}$$

Notice that the lead compensator has its zero to the right of its pole and the lag compensator has its zero to the left of its pole.

The principle behind both compensators is the same. Consider the real singularities (poles and zeros) of the open-loop system. Suppose that the rightmost real singularity is a pole. This open-loop real pole will move towards a real open-loop zero placed to its left when the loop is closed with a positive gain K. If the open-loop system has a pole near $z = 1$, it is usually possible to speed up the closed-loop transient response by adding a zero to its left. For several reasons (the most important will be explained in Section 6 on limitations of control) one should never add just a zero. Thus, one must add a real pole to the left of the added zero, thereby creating a lead compensator. This lead compensator will generally improve the transient response. The best value of the gain K can be determined using the root locus plot of the combined plant and lead compensator.

The lag compensator is used to reduce the steady-state error. This is done by adding a real pole near the point $z = +1$. Adding only a pole will badly slow the closed-loop transient response. Adding a real zero to the left of the pole at $z = 1$ will pull the closed-loop pole to the left for positive gain K, thereby improving the transient response of the closed-loop system.

Another common compensator is the notch filter. It is used when the plant has a pair of lightly damped open-loop poles. These poles can severely limit the range of useful feedback gains, K, because their closed-loop counterparts may become unstable for relatively small values of K. Adding a compensator that has a pair of complex conjugate zeros close to these poles will pull the closed-loop poles towards the zeros as K is increased. One must be careful about the placement of the zeros. If they are placed wrongly, the root locus from the undesirable poles to the added zeros will loop out into the unstable region before returning inside the unit circle. If they are properly placed, this will not happen. Again, one must also add a pair of poles, or the compensator will cause other serious problems, as explained in Section 6.1.

The use of lead and lag compensators is illustrated in the following example.

Design example

Consider a plant with $G(s) = 600/(s + 1)(s + 6)(s + 40)$. This is sampled at $T = 0.0167s$ resulting in $G(z) = 0.000386(z+3.095)(z+0.218)/(z-0.983)(z-0.905)(z - 0.513)$. The root locus for this plant is shown on the left in Fig. 6 as a solid line. Closing the loop with a gain of $K = 1$ results in the closed-loop step response shown at the right as a solid line. The rise time is 0.47, the settling time is 1.35, and the steady-state value is 0.71. There are two aspects of this design that one might want to improve. The step response is rather slow. We would like to make the rise and settling times smaller. The steady-state error in response to a unit step is rather large, 0.29. We would like to make it smaller. Note that increasing the gain from 1 to a larger value

would improve both of these aspects of the step response, but the cost would be a more oscillatory response with a larger overshoot as well as a less robust controller.

A lead compensator, $C_{lead}(z) = K(z - 0.905)/(z - 0.794)$, is added to reduce the rise and settling times without compromising either robustness or overshoot. The zero is placed directly on top of the middle pole of the original plant. The pole is placed so that the largest value of $u(k)$ in the step response of the closed-loop system is less than 4. The resulting root locus is shown as a dotted line in Fig. 6. Closing the loop with $K = 4$ results in the dotted step response shown on the right. The new rise time is 0.267, the settling time is 0.735, and the steady-state value is 0.821. Notice that the lead compensator has improved every aspect of the closed-loop step response. However, the steady-state error in response to a unit step input is still 0.179.

Finally, a lag compensator is added to further reduce the steady-state error in response to a unit step. Adding the lag element makes the complete controller $C_{leadlag}(z) = K(z - 0.905)(z - 0.985)/(z - 0.794)(z - 0.999)$. The pole of the lag compensator is placed close to $z = 1$. The zero is placed just to the right of the pole of the original plant at $z = 0.983$. With these choices, a reasonable gain pulls the added open-loop pole almost onto the added zero. This gives a small steady-state error without significantly compromising the transient response. The new root locus is shown as a dashed line in Fig. 6. The closed-loop step response using this controller is shown dashed at the right of the figure. The rise time is 0.284, the settling time is 0.668, and the steady-state value is 0.986. Note that the steady-state error is now less than 0.02 and the other aspects of the response are nearly as good as they were with only the lead compensator.

5.2 Frequency response-based design

There are two common reasons why one might base a control system design only on the frequency response plots, i.e., on plots of $|G(j\omega)|$ and $\angle G(j\omega)$ versus ω. First, there are systems for which the frequency response can be determined experimentally although an analytical expression for the transfer function is unknown. Although one could estimate a transfer function from this data, it is arguably better not to introduce additional modelling errors by doing this. Second, some systems that are very high order have relatively simple frequency responses. The best example of this is an electronic audio amplifier, which may have approximately 20 energy storage elements. Its transfer function would have denominator degree around 20. Its frequency response plots would be fairly simple, especially since its purpose is to amplify audio signals. In fact, this was the application that drove the work of Bode and Nyquist on feedback. It is also somewhat easier to design a lag compensator in the frequency domain.

One can use either the Bode plots or the Nichols chart of the open-loop system as the basis for the design. Both the gain and phase margin can be

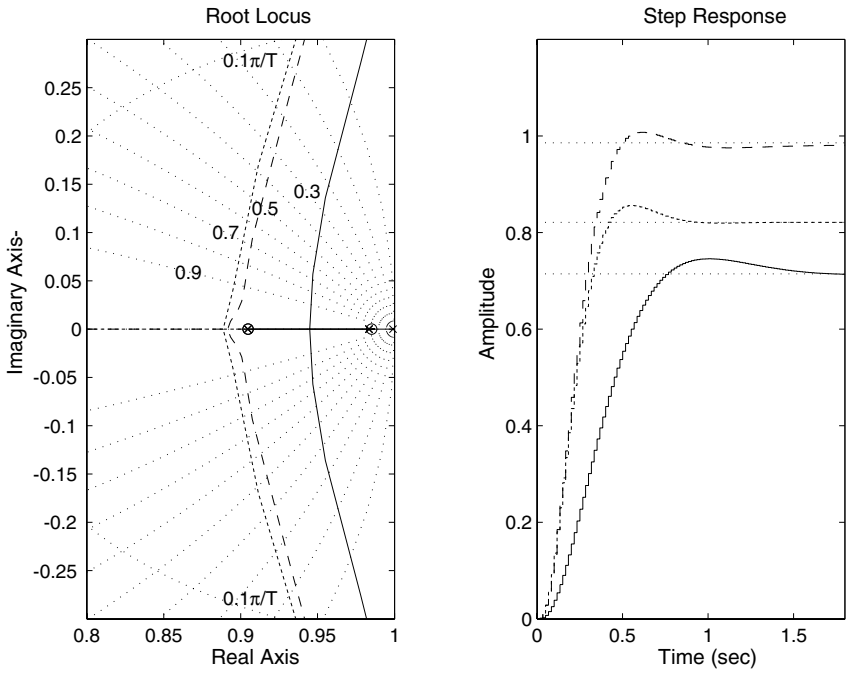


Fig. 6. The root loci and step responses for the design example

read directly from these graphs, making them the most important criteria for design in the frequency domain. There is a convenient relationship between the phase margin and the damping ratio for second-order systems such as (3). It is

$$\zeta \approx (phase\ margin)/100. \tag{14}$$

The effect of a pure gain controller, $C(s) = K$, on the Bode magnitude plot is simply a vertical shift by $20 \log |K|$. The effect on the Nichols chart is a vertical shift by the same amount. Using (14) and the gain and phase margins, the designer can choose a value of the gain K that meets the specifications as in the continuous-time case. If the specifications cannot be satisfied by a pure gain controller, then the various compensators can be tried.

The basic idea behind lead-lag compensation in the frequency domain is that the closed-loop transient response is dominated by the open-loop frequency response near the *gain* and *phase crossover* frequencies, defined to be the frequencies at which the gain crosses 0 dB and the phase crosses $-180°$. The steady-state behavior is determined by the low frequency characteristics of the open-loop frequency response. Thus, the general idea is to add a lag compensator whenever the closed-loop steady-state error is too large. The pole and zero of the lag compensator are placed at low enough frequencies so that they do not affect the open-loop frequency response near the crossover fre-

quencies. On the other hand, the lead compensator is added in the vicinity of the phase crossover frequency where it adds phase margin, thereby improving the transient response as suggested by (14).

The notch filter is used to cancel a large peak in the open-loop frequency response (a resonant peak). The reason it is called a notch filter is evident from its Bode plot. The notch filter has a "notch" in the magnitude of its transfer function. This notch is used to cancel the peak in the open-loop frequency response. The exact placement of the notch is tricky. See [6] for the details.

The design example developed previously using the root locus is repeated below in terms of frequency responses.

Design example revisited

The Bode and Nichols plots for the open-loop plant $G(z) = 0.000386(z + 3.095)(z+0.218)/(z-0.983)(z-0.905)(z-0.513)$ (the same as in Section 5.1), this time with an additional gain of 1.2, are shown in Fig. 7 as solid curves. Closing the loop with unity gain results in a gain margin of 22 dB, a phase margin of 83 degrees, a gain crossover frequency of 2.6 rad/s, and a phase crossover frequency of 14.4 rad/s. Although the gain is slightly higher than it was in our root locus-based design, the closed-loop step response is nearly the same as before, so it is not reproduced here. There is slightly more overshoot and the rise and settling times are slightly faster. We chose the higher gain to emphasize the similarity among the three frequency responses.

The same lead compensator as in the root locus design example is added to speed up the closed-loop response to a unit step. Because of the link between phase margin and damping ratio, ζ (see (14)) we know that increasing the phase margin will speed up the step response. The Bode and Nichols plots of $G(z)C_{lead}(z)$ with a gain of $K = 4$, exactly as in the root locus case, are shown dotted in Fig. 7. Note the slightly more positive phase angle in the critical region near the gain and phase crossover frequencies. The resulting gain margin is 19 dB; the phase margin is 78 degrees; the gain crossover is at 4.25 rad/s; the phase crossover is at 20.4 rad/s. We already know that the resulting closed-loop step response is considerably faster. If we did not know the root locus, we would have placed the maximum phase lead of the lead compensator close to the phase crossover of the original plant.

The same lag compensator as in Section 5.1 is added to reduce the steady-state error in response to a unit step. The Bode and Nichols plots of $G(z)C_{leadlag}(z)$ with a gain of $K = 4$, exactly as in the root locus case, are shown dashed in Fig. 7. The frequency response plots show that the lag compensator greatly increases the DC gain of the open-loop system $(C_{leadlag}(z)G(z))$ while making minimal changes to the frequency response near the critical frequencies. The resulting gain margin is 18 dB; the phase margin is 67 degrees; the gain crossover is at 4.31 rad/s; the phase crossover is at 19.7 rad/s. The lag compensator is placed so that all of its effects occur at lower frequencies than the critical ones.

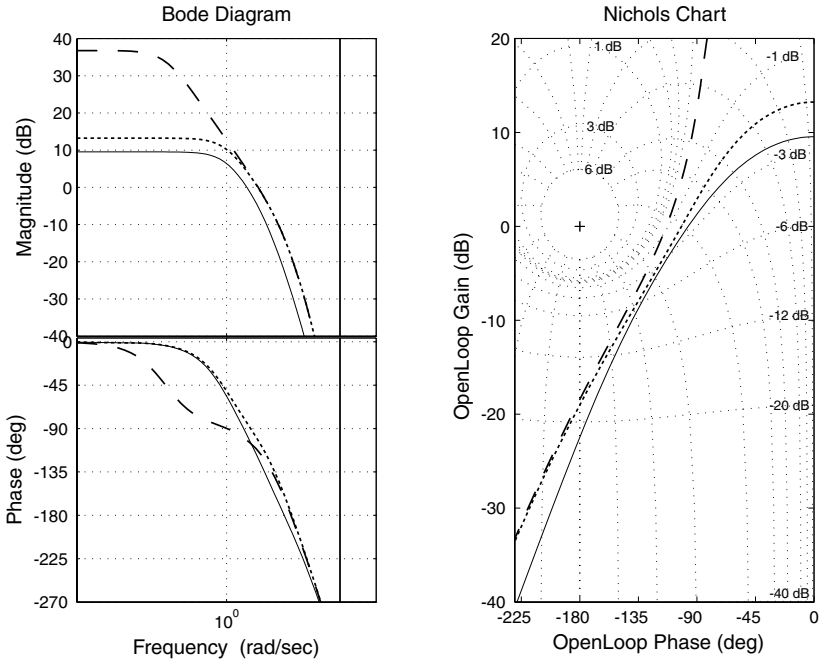


Fig. 7. The Bode and Nichols plots for the design example

5.3 PID control

There are many situations in which it would be inconvenient or impractical to measure the frequency response of the plant and in which the transfer function is either unknown or far too complicated to use for controller design. There are many good examples in the process industries, such as paper-making machines. In many of these applications the specifications are not too demanding. Again, the paper-making machine is illustrative: the transient response is not very important, but tight steady-state control of the thickness is. This is the paradigmatic use of PID control, although the method is also used for much more demanding applications, including many for which a good plant model is known.

The discrete-time (proportional + integral + derivative) (PID) controller is derived from the original continuous-time version. A realistic, as opposed to academic, version of the continuous-time PID controller is

$$C_{PID}(s) = K_P + K_I \frac{1}{s} + K_D \frac{s}{1 + sT_f}. \tag{15}$$

It is also common in practice to use $(\alpha R(s) - Y(s))$ as the input to the derivative term (coefficient K_D) instead of $(R(s) - Y(s))$. Often, α is set to zero. The continuous-time controller can be discretized in a variety of ways, each with

its advantages and disadvantages; see [1]. The most commonly used method is the backwards difference method, which is known to be well behaved. The result can be written as

$$u(k) = u_P(k) + u_I(k) + u_D(k), \tag{16}$$

where

$$u_P(k) = K_P(r(k) - y(k)) \tag{17}$$

$$u_I(k) = u_I(k-1) + K_I T(r(k) - y(k)) \tag{18}$$

$$u_D(k) = (1 + \frac{T}{T_f})^{-1} u_D(k-1) - \frac{K_D}{T_f}(\frac{1}{1 + \frac{T}{T_f}})(y(k) - y(k-1)), \tag{19}$$

where T is the sampling interval and T_f is a filtering coefficient.

One can purchase a PID controller as an essentially turnkey device. It is connected to the plant and the 3–5 parameters of the controller are then adjusted (tuned) to the particular plant. There is an extensive literature on tuning PID controllers dating back at least to Ziegler and Nichols [1]. The basic ideas are relatively simple if one sets the D-terms to zero. One straightforward tuning method is to set the D- and I-terms to zero and gradually increase the gain K_P just until the closed-loop step response becomes unstable. Reducing the gain by 50%, for example, will produce a closed-loop system with a gain margin of 6 dB. For a proportional controller this is regarded as a fairly good choice of gain. If this is sufficient to meet the specifications, there is no more to be done.

If the steady-state error is too large, an I-term must be added to the controller. Doing so adds a pole at $z = +1$, thereby eliminating the steady-state error in response to a unit step. It will also add a zero at $-K_I/K_P$ in the continuous-time case. Some modest fine-tuning of the two gains will improve the transient response without, of course, changing the steady-state error. The well-known but overly aggressive Ziegler–Nichols rules suggest decreasing K_P to 40% of the value of K_P that caused instability and then choosing $K_I = K_P/(0.8T_u)$, where T_u is the period of the oscillation that resulted when K_P was chosen to make the closed-loop system unstable.

Tuning the D-term is notoriously difficult. Its basic role is to add a zero and a pole to the controller. If chosen properly this zero and pole will act as a lead compensator and speed up the closed-loop transient response. See [1] for details.

If one has a good mathematical description of the plant, then either a root locus plot, a Bode plot, or a Nichols chart of the open-loop system can be used to choose the parameters of the PID controller (which is basically a lead-lag controller with the lag pole at $z = 1$) to achieve a desired step response.

It is now possible to buy "self-tuning" PID controllers. They are available from several manufacturers and they use a variety of tuning methods. The details are often proprietary. Generally, an operator commands the controller

to enter its tuning mode. The controller then tunes itself and switches to operate mode and stays there until it is again commanded to tune itself.

5.4 Design by emulation

In the discussion above, we have described the basics of *direct digital design*, meaning that the plant is first discretized (taking into account the effects of sampling and ZOH) and a discrete-time controller is designed.

An alternative is to initially ignore the effects of D/A and A/D conversion and design a continuous compensator $C(s)$ for the continuous time plant $G(s)$. The continuous-time compensator is then discretized to obtain $C_d(z)$. This procedure, known as *design by emulation*, may be used when a working continuous-time controller already exists or when the designer has very good intuition for continuous-time control.

The conversion of a continuous-time controller to an approximately equivalent discrete-time controller can be done in a variety of ways. Two simple and useful methods require only that s in the continuous-time controller be replaced by the appropriate formula involving z. They are:

- Backward rule: $s = \frac{(z-1)}{T_s}$
- Tustin's method: $s = \frac{2}{T_s}\frac{(z-1)}{(z+1)}$.

A third method is only slightly more complicated.

- Matched pole-zero (MPZ) method:

Recall that the poles of a continuous-time transfer function $C(s)$ are related to the poles of its z-transform $C_d(z) = \mathcal{Z}\{C(s)\}(z)$ by

$$z = e^{sT},$$

where T is the sampling period. One can then attempt to obtain a digital version of $C(s)$ by applying this relationship to its zeros as well as its poles (we stress that this represents only an approximation—the zeros are not related by $z = e^{sT}$). The resulting discrete-time transfer function $C_d(z)$ is then obtained with a minimum of calculations.

If $C(s)$ is strictly proper (the degree of its denominator is greater than that of its numerator) it is sometimes desirable to further modify the resulting $C(z)$ by multiplying it repeatedly by $(1+z^{-1})$ (adding zeros at $z=-1$) until the resulting transfer function has denominator degree equal to that of the numerator, or equal to that of the numerator minus one ("modified matched pole-zero method"). Doing so has the effect of "averaging" past and present inputs. The MPZ method requires inputs of up to $e(k+1)$ in order to produce $u(k+1)$. This may be undesirable in applications where the time to compute $u(k+1)$ is significant compared with the sampling period. The modified MPZ method does not suffer from this drawback, as it requires only "past" inputs to produce the current output.

The approximations obtained via the methods described here are typically useful at frequencies below 1/4 of the sampling rate. Furthermore, because design by emulation ignores the effect of the ZOH, the performance of the resulting controllers yields reasonable results at sampling rates that are approximately 20 times or higher than the bandwidth of the continuous-time plant. For lower sampling rates, it is important to analyze the resulting closed-loop system in discrete time to ensure adequate performance.

5.5 Advanced methods of control

One method of control design is only slightly more involved than those discussed so far and lends itself very well to digital implementation. It is known as the two-degrees-of-freedom (2DOF) method. The basic idea is to divide the controller into two nearly independent components as shown in Fig. 8. The

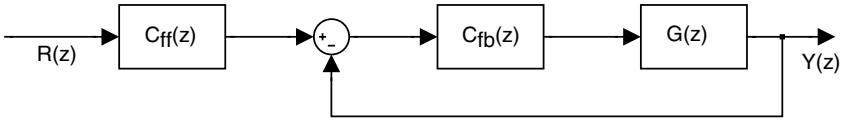


Fig. 8. A 2DOF controller

feedback component of the controller, $C_{fb}(z)$, is designed to deal primarily with disturbances while the feedforward component $C_{ff}(z)$ deals mainly with the response to a command signal $R(z)$. Although $C_{ff}(z)$ acts open loop, it can be realized very accurately on the computer. Thus, there should be minimal uncertainty associated with its behavior. The feedback portion of the controller, $C_{fb}(z)$, is designed to minimize the effects of plant uncertainty and to make the closed-loop system have a gain of one within the frequency range of possible inputs.

There is a very large literature on controller design. There are state-space methods for arbitrarily placing the poles of the closed-loop system assuming only that the open-loop system is controllable and observable [3]. Because it is not at all obvious where the closed-loop poles should be placed, there is also a large literature on optimal control. For linear systems, the linear quadratic regulator and the H_2 and H_∞ methods are particularly important [15]. There has also been much research and some applications in the field of nonlinear control. Introductions to all of these topics can be found in [9].

6 Limitations on Control Systems

It is very important for the control system designer to be aware of several limitations on the stability and performance of real control systems. These limitations are due to inaccuracies in the plant model, inevitable disturbances that

enter the system, actuator saturation, and fundamental unavoidable trade-offs in the design. The first step in appreciating these limitations is to examine the more realistic picture of a SISO control loop in Fig. 9, where $G_n(z)$ is the

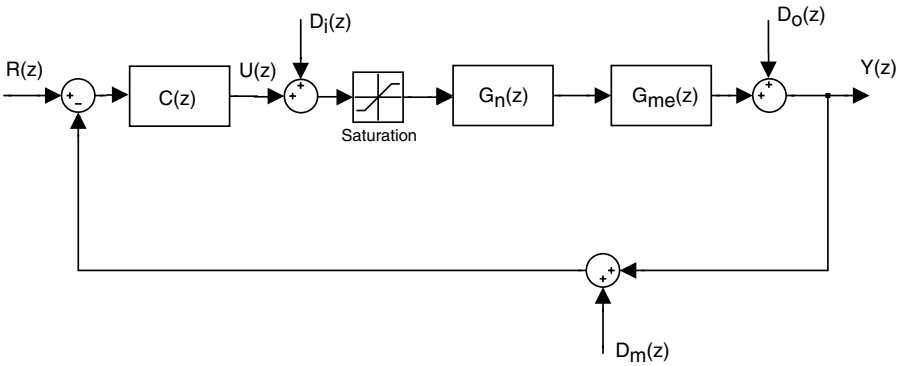


Fig. 9. A feedback control system with input, output, and measurement disturbances

nominal plant which is used in the design. The actual plant is $G_{me}(z)G_n(z)$ where $G_{me}(z) = (1+G_{\Delta}(z))$ denotes modelling errors. Generally, only bounds are known for the multiplicative modelling error, $G_{\Delta}(z)$. In particular, in a networked and embedded control system the phase of $G_{\Delta}(z)$ is known only to lie within limits determined by the timing accuracy of the system. The additional inputs are $D_i(z)$ representing input disturbances, $D_o(z)$ for output disturbances, and $D_m(z)$ for measurement noise. Note that we have omitted any sensor dynamics in order to focus on the most essential aspects of robustness and sensitivity.

LTI control systems are also limited by the fundamental Bode gain-phase relation. The precise theorem can be found in [14]. A simple rule of thumb based on Bode's result is that each $-20n$ dB/decade of reduction in the open-loop gain implies $\approx -90°n$ of phase shift, where n is a positive integer. This link between gain and phase is easily seen in lead and lag compensators. A lead compensator basically adds positive phase to improve the transient performance of the closed-loop system. The price paid for this positive phase is an undesirable increase in high frequency gain. A lag compensator is used to add to the DC gain of the open-loop system, thus decreasing the steady-state error of the closed-loop system. The price paid for this improvement is an undesirable negative phase shift.

6.1 Sensitivity to disturbances

The effect of the disturbances on the performance of the control system can be studied by writing the transfer functions from each disturbance to $Y(z)$

and $U(z)$. The effect of the disturbances on $U(z)$ is particularly important because of saturation. The transfer functions are written using the nominal plant so they are nominal sensitivities.

$$\begin{aligned}
\frac{Y(z)}{R(z)} &= G_{cl}(z) = \frac{G_n(z)C(z)}{1+G_n(z)C(z)} \\
\frac{Y(z)}{D_i(z)} &= S_{io}(z) = \frac{G_n(z)}{1+G_n(z)C(z)} \\
\frac{Y(z)}{D_o(z)} &= S_o(z) = \frac{1}{1+G_n(z)C(z)} \\
\frac{Y(z)}{D_m(z)} &= -G_{cl}(z) = -\frac{G_n(z)C(z)}{1+G_n(z)C(z)} \\
\frac{U(z)}{D_m(z)} &= S_{ou}(z) = \frac{C(z)}{1+G_n(z)C(z)}
\end{aligned} \tag{20}$$

Notice that $S_{ou}(z)$ is also the transfer function from $R(z)$ and $D_o(z)$ to $U(z)$, which explains why it is a bad idea to use a zero as a lead compensator without also including a pole. Such a choice would result in $C(z) = (z - z_l)$, and this would cause $S_{ou}(z)$ to amplify any high frequency components of $R(z)$, $D_o(z)$, and $D_m(z)$. This would result in actuator saturation on noise. Ultimately, the placement of the pole in a lead compensator and hence, the amount of phase lead possible is limited by the amplitude of the disturbances.

A fundamental limit on controller performance is easily derived from the transfer functions above:

$$G_{cl}(z) + S_o(z) = 1, \quad \text{for all } z \in \mathbb{C}. \tag{21}$$

Another limitation follows from the fact that $G_{cl}(z)$ is the transfer function from both $-D_m(z)$ and $R(z)$ to $Y(z)$. This makes it very desirable to keep $|C(z)|$ small at those frequencies at which $R(z)$ is zero. A typical example is in aircraft SCAS where pilot inputs and aircraft maneuvers are known to be limited to relatively low frequencies, implying that any signal at high frequency must be noise. Now consider the implications of (21) for a closed-loop system having the property that $|G_{cl}(z)|$ is small at high frequency. Such a system will pass output disturbances at those frequencies without attenuation.

6.2 Robustness

It is important that the closed-loop system remain stable despite the differences between the nominal plant used for the controller design and the real plant. There has been extensive research devoted to robust stability in recent years. There are many results available; see [9, 15]. The following is a simple example from [8].

Theorem 1. *Consider a plant with nominal transfer function $G_n(z)$ and actual transfer function $G_n(z)(1 + G_\Delta(z))$. Assume that $C(z)$ is a controller that achieves internal stability for $G_n(z)$. Assume also that $G_n(z)C(z)$ and $G_n(z)(1 + G_\Delta(z))C(z)$ both have the same number of unstable poles. Then a sufficient condition for stability of the true feedback loop obtained by applying the controller $C(z)$ to the actual plant is that*

$$|G_{cl}(z)||G_\Delta(z)| = \left| \frac{G_n(z)C(z)}{1 + G_n(z)C(z)} \right| |G_\Delta(z)| < 1. \tag{22}$$

The proof is a straightforward application of the Nyquist stability theorem. Notice that the theorem holds regardless of the uncertainties in the phase. Thus, it is valuable in ensuring that delays due to networking and computing cannot compromise the stability of the real closed-loop system.

The use of the theorem can be understood by dividing the frequency response of the nominal open-loop system and compensator, $G_n(z)C(z)$, into three regions. In the low frequency region, it is normally true that $|G_\Delta(z)|$ is small. Thus, the controller can have high gain and the nominal closed-loop system can have a magnitude near one without compromising stability robustness. At high frequencies, $|G_\Delta(z)|$ is usually large but $|G_n(z)C(z)|$ is small. Again, stability robustness is not a problem because the nominal closed-loop system has small magnitude. The critical region is the frequency range near the gain and phase crossover frequencies. In this region, the bounds on $|G_\Delta(z)|$ are very important.

6.3 Trade-offs

The following theorem [14], due originally to Bode [4], proves that there is a fundamental trade-off inherent in any attempt to reduce the sensitivity, $S_o(z)$, of a closed-loop system.

Theorem 2. *Consider a SISO LTI discrete-time open-loop system $G_n(z)C(z)$ with its corresponding stable closed-loop system $G_{cl}(z) = \frac{G_n(z)C(z)}{1+G_n(z)C(z)}$ and sensitivity $S_o(z) = \frac{1}{1+G_n(z)C(z)}$. Then*

$$\int_{-\pi}^{\pi} \ln |S_o(e^{j\omega})| d\omega = 2\pi \sum_i (\ln |p_i| - \ln |\gamma + 1|) \tag{23}$$

where the p_i are the unstable poles of the open-loop system and $\gamma = \lim_{z\to\infty} G_n(z)C(z)$.

Notice that if the open-loop system is stable and strictly proper, then the theorem implies that $\int_{-\pi}^{\pi} \ln |S_o(e^{j\omega})| d\omega = 0$. Typically, one wants to design the controller to keep the sensitivity small at low frequencies. The theorem proves that the inevitable consequence is that the controller increases the sensitivity at high frequencies.

7 Beyond This Introduction

There are many good textbooks on classical control. Two popular examples are [5] and [6]. A less typical and interesting alternative is the recent textbook [8]. All three of these books have at least one chapter devoted to the basics of digital control. Textbooks devoted to digital control are less common, but there are some available. The best known is probably [7]. Other possibilities are [2, 12, 13] An excellent book about PID control is the one by Aström and Hägglund [1]. Good references on the limitations of control are [10] and [11]. Bode's book [4] is still interesting, although the emphasis is on vacuum tube circuits.

References

1. K. J. Åström and T. Hägglund. *PID Controllers: Theory, Design and Tuning*. International Society for Measurement and Control, Seattle, WA, 2nd edition, 1995.
2. K. J. Åström and B. Wittenmark. *Computer Controlled Systems*. Prentice-Hall, Englewood Cliffs, NJ, 3rd edition, 1996.
3. P. R. Bélanger. *Control Engineering: A Modern Approach*. Saunders College Publishing, Stamford, CT, 1995.
4. H. W. Bode. *Network Analysis and Feedback Amplifier Design*. R. E. Krieger Pub. Co., Huntington, NY, 1975.
5. R. C. Dorf and R. H Bishop. *Modern Control Systems*. Prentice-Hall, Upper Saddle River, NJ, 10th edition, 2004.
6. G. F. Franklin, J. D. Powell, and A. Emami-Naeini. *Feedback Control of Dynamical Systems*. Prentice-Hall, Upper Saddle River, NJ, 4th edition, 2002.
7. G. F. Franklin, J. D. Powell, and M. L. Workman. *Digital Control of Dynamic Systems*. Prentice-Hall, Upper Saddle River, NJ, 3rd edition, 1997.
8. G. C. Goodwin, S. F. Graebe, and M. E. Salgado. *Control System Design*. Prentice-Hall, Upper Saddle River, NJ, 2000.
9. W. S. Levine, editor. *The Control Handbook*. CRC Press, Boca Raton, FL, 1996.
10. D. P. Looze and J. S. Freudenberg. *Frequency Domain Properties of Scalar and Multivariable Feedback Systems*. Springer-Verlag, Berlin, 1988.
11. D. P. Looze and J. S. Freudenberg. Tradeoffs and limitations in feedback systems. In W. S. Levine, editor, *The Control Handbook*, pages pp. 537–550. CRC Press, Boca Raton, FL, 1996.
12. M. S. Santina and A. R. Stubberud. *Discrete-Time Equivalents to Continuous-Time Systems*. Eolss Publishers Co. Ltd., Oxford, U.K., 2004.
13. M. S. Santina, A. R. Stubberud, and G. H. Hostetter. *Digital Control System Design*. International Thomson Publishing, Stamford, CT, 2nd edition, 1994.
14. B.-F. Wu and E. A. Jonckheere. A simplified approach to Bode's theorem for continuous-time and discrete-time systems. *IEEE Transactions on Automatic Control*, 37(11):1797–1802, Nov. 1992.
15. K. Zhou, J. C. Doyle, and K. Glover. *Robust and Optimal Control*. Prentice-Hall, Upper Saddle River, NJ, 1995.

Basics of Sampling and Quantization

Mohammed S. Santina[1] and Allen R. Stubberud[2]

[1] The Boeing Company, Seal Beach, CA, 90740, U.S.A.
[2] University of California, Irvine, CA, 92697-2625, U.S.A.

1 Introduction

The controller in an analog control system uses analog electronic, mechanical, electromechanical, or hydraulic devices. In contrast, a digital control system uses digital electronics hardware, usually in the form of a programmed digital computer, as the heart of the controller. Like analog controllers, digital controllers normally have analog elements at their periphery to interface with the (analog) plant; thus, it is the internal workings of the controller that distinguishes a digital control system from an analog control system. As a result of using digital computers as control system controllers, the signals in the system controller must be in the form of digital signals, and the control system itself usually is treated mathematically as a discrete-time system. In this chapter, the two operations, the sampling of continuous-time signals and the reconstruction of a continuous-time signal from samples, are considered. The sampling rate is an important parameter in the design of a digital control system. The best sampling rate for a digital control system is the lowest rate that meets all performance requirements. Selection of the sampling rate to meet certain performance requirements is discussed. Because digital controllers are implemented with finite word length registers and finite precision arithmetic, their signals and coefficients can attain only discrete values. Therefore, further analysis is needed to determine if the performance of the resulting digital controller in the presence of signal and coefficient quantization is acceptable. In the final section of the chapter, we discuss error sources that exist in the digital signal processing that takes place in digital controllers. These error sources are generated by coefficient quantization, by quantization in analog-to-digital conversion, and by arithmetic operations. Limit cycles and deadbands are also discussed.

1.1 Sampling and the sampling theorem

Sampling is the process of deriving a discrete-time sequence from a continuous-time function. Usually, but not always, the samples are evenly spaced in time. *Reconstruction* is the inverse of sampling; it is the formation of a continuous-time function from a sequence of samples. Many different continuous-time functions can have the same set of samples, so that, except in highly restricted circumstances, a sampled function is not uniquely determined by its samples. A signal $g(t)$ and its Fourier transform $G(\omega)$ are generally related by

$$G(\omega) = \int_{-\infty}^{\infty} g(t)e^{-j\omega t}dt \tag{1}$$

$$g(t) = \frac{1}{2\pi}\int_{-\infty}^{\infty} G(\omega)e^{j\omega t}d\omega. \tag{2}$$

This relationship is similar to the (one-sided) Laplace transformation with $s = j\omega$, except that the transform integral of (1) extends over all time rather than from $t = 0^-$ to $t = \infty$ and the region of convergence for the Laplace transform might not include the line $s = j\omega$. The Fourier transform $G(\omega)$ is termed the *spectrum* of $g(t)$.

If a signal $g(t)$ is uniformly sampled with sampling period (interval) T to form the sequence

$$g(kT) = g(t = kT),$$

then the corresponding impulse train that extends from the time origin both ways in time is

$$g^*(t) = \sum_{k=-\infty}^{\infty} g(kT)\delta(t - kT), \tag{3}$$

which is a continuous-time signal, equivalent to $g(kT)$, and has the Fourier transform

$$G^*(\omega) = \frac{1}{T}\sum_{n=-\infty}^{\infty} G(\omega - n\omega_s), \tag{4}$$

where

$$\omega_s = 2\pi f_s = 2\pi/T.$$

The proof of this result is well documented and can be found in [1]. The function $G^*(\omega)$ in (4) is periodic in ω, and each individual term in the series has the same form as the original $G(\omega)$, with the exception that the nth term is centered at

$$\omega = n(2\pi/T), \quad n = \ldots, -2, -1, 0, 1, 2, \ldots.$$

In general, then, if $G(\omega)$ is not limited to a finite frequency range, these terms overlap each other along the ω-axis. One important situation in which samples

of a continuous-time signal $g(t)$ are unique occurs when its Fourier transform $G(\omega)$ is *bandlimited* . A signal is bandlimited at (hertz) frequency f_B if

$$G(\omega) = 0 \quad \text{for} \quad |\omega| > 2\pi f_B = \omega_B.$$

In this case, (2) can be rewritten

$$g(t) = \frac{1}{2\pi} \int_{-\omega_B}^{\omega_B} G(\omega) e^{j\omega t} d\omega. \tag{5}$$

If the sampling frequency f is more than twice the bandlimit frequency f_B, the individual terms in (4) do not overlap with $G(\omega)$ and thus $g(t)$ can be determined from $G^*(\omega)$, which in turn, is determined from the samples $g(k)$. The smallest sampling frequency f for which the individual terms in (4) do not overlap is exactly twice the bandlimit frequency f_B. The frequency $2f_B$ is termed the *Nyquist frequency* for a bandlimited signal. If the sampling frequency does not exceed the Nyquist frequency, the individual terms in (4) overlap, a phenomenon called *aliasing* (or *foldover*). Note that the radian sampling frequency is given by $\omega_S = 2\omega_B$ and, therefore,

$$T = 2\pi/\omega_S = \pi/\omega_B = \frac{1}{2f_B}.$$

which relates the sampling period to the highest frequency f_B in the signal. The above results are summarized in the statement of the sampling theorem as:

The uniform samples of a signal $g(t)$ that is bandlimited above (hertz) frequency f_B are unique if and only if the sampling frequency is higher than $2f_B$. That is, in terms of the sampling period, aliasing will not occur if

$$T < \frac{1}{2f_B}. \tag{6}$$

It is apparent from the above discussion that the sampling period T in a digital control system is an important design parameter which must be chosen appropriately for a digital control system to function properly.

1.2 Analog-to-digital conversion

Digital control system analysis and design methods are usually presented as if the controller signals are continuous-amplitude signals, that is, they can take on a continuum of values. However, because digital controllers are implemented with finite word length registers and finite precision arithmetic, their signals and coefficients can attain only discrete values, that is, they are *quantized* . A signal consisting of quantized samples is called a *digital signal*. A device which samples a signal and then quantizes the samples is called an *analog-to-digital* (A/D) converter. An A/D converter produces a binary

representation, using a finite number of bits, of the sampled signal at each sample time. Using a finite number of bits to represent a signal sample results in *quantization errors* in the A/D process. For example, the maximum quantization error in a 16-bit A/D conversion is $2^{-16} = 0.0015\%$, which is very low compared to typical errors in analog signals. This error, if taken to be "noise," gives a *signal-to-noise ratio* (SNR) of $20\log_{10}(2^{-16}) = 96.3$ dB which is much better than what is required of most control systems. The control system designer must ensure that enough bits are used to give the control system its desired accuracy. The effects of roundoff and truncation errors in digital computation are discussed later in this chapter along with the importance of using adequate word lengths in fixed- or floating-point computations. Although minimizing word length is not as important today as it was in the past when digital hardware was very expensive, it is still an important design consideration.

1.3 Reconstruction and digital-to-analog conversion

Reconstruction is the formation of a continuous-time function from a sequence of samples. Many different continuous-time functions can have the same set of samples, so a reconstruction is not unique. Reconstruction is performed using *digital-to-analog* (D/A) converters. Electronic D/A converters typically produce a step reconstruction from incoming signal samples by converting the binary-coded digital input to a voltage, transferring the voltage to the output, and holding the output voltage constant until the next sample is available. The operation of holding each of the samples $f(k)$ for a sampling interval T to form a step reconstruction is called *sample and hold* (S/H). The resultant continuous-time function generated by the step reconstruction is denoted by $f^0(t)$. The step reconstruction of a continuous-time signal from samples can be represented as a two-step process: (a) converting the sequence $f(k)$ to its corresponding impulse train $f^*(t)$, where

$$f^*(t) = \sum_0^\infty f(k)\delta(t - kT), \tag{7}$$

and (b) integrating the impulse train which results in the step reconstruction signal $f^0(t)$. This viewpoint neatly separates the two steps of reconstruction, the conversion of the discrete sequence to a continuous-time waveform and the details of the shaping of the reconstructed waveform. The continuous-time transfer function that converts the impulse train with sampling interval T to a step reconstruction is termed a *zero-order hold* (ZOH). Each incoming impulse in (7) to the ZOH produces a rectangular pulse of duration T. Therefore, the transfer function of the ZOH is given by

$$L_0(s) = \frac{1}{s}(1 - e^{-sT}). \tag{8}$$

The accuracy of the reconstruction can be improved by employing a hold of higher order than the ZOH. An nth-order hold produces a piecewise nth degree polynomial that passes through the most recent $n+1$ input samples. It can be shown that, as the order of the hold is increased, a well-behaved signal is reconstructed with increased accuracy. Higher order holds do, however, introduce significant time lags that can have a major negative effect on the stability of any closed loop systems in which they are embedded.

1.4 Discrete-time equivalents of continuous-time systems

When a digital controller is designed to control a continuous-time plant it is important to have a good understanding of the plant to be controlled as well as the controller and its interfaces with the plant. There are two fundamental approaches to designing discrete-time control systems for continuous-time plants. The first approach is to derive a discrete-time equivalent of the plant and then directly design a discrete-time controller to control the discretized plant. This approach to designing a digital controller directly parallels the classical approach to analog controller design. The other approach to designing discrete-time control systems for continuous-time plants is to first design a continuous-time controller for the plant, and then to derive a digital controller that closely approximates the behavior of the original analog controller. The controller design can approximate the integrations of the continuous-time controller with discrete-time operations or it can be made to have step (or other) response samples that are equal to samples of the analog controller's step (or other) response. Usually, however, even for small sampling periods, the discrete-time approximation does not perform as well as the continuous-time controller from which it was derived. Discrete-time equivalents of continuous-time systems are discussed in great detail in [2].

2 Sample-Rate Selection

It is apparent from the above discussion that the sampling rate is a critical design parameter in the design of digital control systems. Usually, as the sampling rate is increased, the performance of a digital control system improves; however, computer costs also increase because less time is available to process the controller equations, and thus higher performance computers must be used. Additionally, for systems with A/D converters, higher sample rates require faster A/D conversion speed which may also increase system costs. Reducing the sample rate for the sake of reducing cost, on the other hand, may degrade system performance or even cause instability. Aside from cost, the selection of sampling rates for digital control systems depends on many factors. Some of these factors include smoothness of the time response, effects of disturbances and sensor noise, parameter variations, and quantization. Selection of the sampling interval also depends on the reconstruction method

used to recover the bandlimited signal from its samples [1]. Another statement of the sampling theorem which is related to signal reconstruction states that *when a bandlimited continuous-time signal is sampled at a rate higher than twice the bandlimit frequency, the samples can be used to reconstruct uniquely the original continuous-time signal. In general, the best sampling rate which can be chosen for a digital control system is the slowest rate that meets all performance requirements.* Although the sampling theorem is not directly applicable to most discrete-time control systems because typical input signals (e.g., steps and ramps) are not bandlimited and because good reconstruction requires long time delays, it does provide some guidance in selecting the sample rate and also in deciding how best to filter sensor signals before sampling them.

Åström and Wittenmark [3] suggest, by way of example, a criterion for the selection of the sample rate that depends on the magnitude of the error between the original signal and the reconstructed signal. The error decreases as the sampling rate is increased considerably higher than the Nyquist rate. Depending on the hold device used for reconstruction, the number of samples required may be several hundreds per Nyquist sampling period.

2.1 Control system response and the sampling period

The main objective of many digital control system designs is to select a controller so that the system-tracking output, as nearly as possible, tracks or "follows" the tracking command input. Usually, the first figure of merit that the designer selects is the closed loop bandwidth, f_c (Hz), of the feedback system because f_c is related to the speed at which the feedback system can track the command input. Also, the bandwidth f_c is related to the amount of attenuation the feedback system must provide in the face of plant disturbances. It is then appropriate to relate the sampling period to the bandwidth f_c as suggested by the sampling theorem because the bandwidth of the closed loop system is related to the highest frequency of interest in the command input. As a general rule, the sampling period should be chosen in the range

$$\frac{1}{30f_c} < T < \frac{1}{5f_c}. \tag{9}$$

Of course, other design requirements may require even higher sample rates, but sampling rates less than 5 times f_c are usually not desirable and should be avoided if possible.

Another criterion for selecting the sampling period is based on the *rise time* of the feedback system so as to provide smoothness in the time response. It can be shown that the rise time (10% to 90%), T_r, of a first-order system of the form

$$H(s) = \frac{1}{\tau s + 1}$$

is given by

$$T_r = 2.2\tau.$$

The sampling period, in terms of the rise time, can be selected according to

$$0.095T_r < T < 0.57T_r, \tag{10}$$

which is derived from (9). Similarly, the rise time of the canonical second-order system

$$H(s) = \frac{\omega_n^2}{s^2 + 2\zeta\omega_n s + \omega_n^2}$$

is $T_r = (\pi - \beta)/\omega_d$ where $\omega_d = \omega_n\sqrt{1-\zeta^2}$ and $\beta = \sin^{-1}\left(\sqrt{1-\zeta^2}\right)$. For a damping ratio $\zeta = 0.707$, the rise time is $T_r = 3.33/\omega_n$. Based on (9), the sampling period is given by

$$0.06T_r < T < 0.4T_r. \tag{11}$$

In digital control systems, a time delay of up to a full sample period is possible before the digital controller can respond to the next input command; therefore, Franklin et al. [4] suggest that the time delay be kept to about 10% of the rise time, which suggests that the sampling period should satisfy

$$T < 0.05/f_c. \tag{12}$$

Another criterion for selecting the sample period, which depends on the frequency response of the continuous-time system, is given by Åström and Wittenmark [3]. The sampling rate is selected such that

$$0.15 < T\omega_0 < 0.5, \tag{13}$$

where ω_0 is the gain crossover frequency of the continuous-time system in radians per second. A detailed discussion on sampling rate selection is found in [5].

3 Quantization Effects

Digital controllers are implemented with finite word length registers and finite precision arithmetic; therefore, their signals and coefficients can attain only discrete values. Further analysis is thus needed to determine if the performance of a resulting digital controller in the presence of signal and coefficient quantization is acceptable [6]. In this section, three error sources that may occur in the digital processing performed by digital controllers are discussed. These error sources are (a) coefficient quantization, (b) quantization in A/D conversion of signals, and (c) arithmetic operations on quantized signals and coefficients. *Limit cycles* and *deadbands* are also discussed very briefly. Before discussing these errors, however, a brief review of fixed- and floating-point number arithmetic is presented.

3.1 Fixed-point and floating-point number representations

There are many choices of arithmetic that can be used to implement digital controllers. The two most popular ones are *fixed-point* and *floating-point* binary arithmetic. Other non-standard arithmetic such as *logarithmic* and *residue* representations [7] are also possibilities but are not discussed here.

Fixed-Point Arithmetic

In general, an n-bit fixed-point binary number N can be expressed as

$$\begin{aligned} N &= \sum_{j=-m}^{n-1} b_j 2^j = b_{n-1}2^{n-1} + b_{n-2}2^{n-2} + \cdots + b_1 2^1 + b_0 2^0 \\ &\quad + b_{-1}2^{-1} + b_{-2}2^{-2} + \cdots + b_{-m}2^{-m} \\ &= (b_{n-1} \cdots b_0 \bullet b_{-1} b_{-2} \cdots b_{-m})_2, \end{aligned} \tag{14}$$

where b_j can be either a zero or a one. The bit b_{n-1} is termed the *most significant bit* (MSB) and $b_{-m}$is termed the *least significant bit* (LSB). The integer portion of the number, $b_n b_{n-1} \cdots b_0$, is separated from the fractional portion, $b_{-1} b_{-2} \cdots b_{-m}$, by the *binary point* or radix point. For example, the binary number 1101.101 has the decimal value

$$1101.101 = 1(2^3) + 1(2^2) + 0(2^1) + 1(2^0) + 1(2^{-1}) + 0(2^{-2}) + 1(2^{-3}) = 13.625$$

In fixed-point arithmetic, numbers are always normalized to be binary fractions (i.e., less than one) of the form

$$b_0 \bullet b_1 b_2 \cdots b_C,$$

where b_0 is the *sign bit*. The $(C+1)$-bit normalized number is stored in a register with the sign bit separated from the C-bit number by a *fictitious* binary point. The binary point is fictitious because it does not occupy any bit location in the register. The *word length*, C_l, is defined as the number of bit locations in the register to the right of the binary point.

There are three commonly used methods for representing signed numbers: (a) signed-magnitude, (b) two's complement, and (c) one's complement. Consider the $(C+1)$-bit binary fraction $b_0 \bullet b_1 b_2 \cdots b_C$ where b_0 is the sign bit. In the *signed-magnitude* representation, the fractional number is positive if b_0 is zero and it is negative if b_0 is one. For example, the decimal number 0.75 equals 0.11 in binary signed-magnitude representation, and -0.75 equals 1.11. In a signed-magnitude representation, binary numbers can be converted to decimal numbers using the relationship

$$N = (-1)^{b_0} \sum_{i=1}^{C} b_i 2^{-i}. \tag{15}$$

The *two's complement* representation of a positive number is the same as the signed-magnitude representation. A two's complement representation of a

negative number, however, is obtained by *complementing* (i.e., replacing every 1 with 0 and every 0 with 1) all the bits of the positive number and adding one to the LSB of the complemented number. For example, the two's complement representation of the decimal number 0.75 is 0.11 and the two's complement representation of -0.75 is 1.01. A decimal number can be recovered from its two's complement representation using the relationship

$$N = -b_0 + \sum_{i=1}^{C} b_i 2^{-i}. \tag{16}$$

The one's complement representation of fractional numbers is the same as the two's complement without the addition of one to the LSB. For example, the one's complement representation of 0.75 is 0.11 and the one's complement representation of -0.75 is 1.00. A decimal number can be recovered from its one's complement representation via the relationship

$$N = b_0 \left(2^{-C} - 1\right) + \sum_{i=1}^{C} b_i 2^{-i}. \tag{17}$$

The two's complement representation of binary numbers has several advantages over the signed-magnitude and the one's complement representations [8], and therefore it is more popular. In general, the sum of two normalized C-bit numbers using fixed-point arithmetic is a C-bit number while the product of two C-bit numbers is a $2C$-bit number. Hence, if the register word length is fixed to C bits, a *quantization error* is introduced in multiplication but not in addition.[3] The product is quantized either by *rounding* or by *truncation*. For example, rounding the binary number 0.010101 to four bits after the binary point gives 0.0101 but rounding it to three bits yields 0.011. When a number is truncated, all the bits to the right of its LSB are discarded. For example, truncating the number 0.010101 to three bits after the binary point gives 0.010.

Floating-Point Arithmetic

A major disadvantage of fixed-point arithmetic is the limited range of numbers that can be represented with a given word length. Another type of arithmetic which, for the same number of bits, has a much larger range of numbers is *floating-point arithmetic*. In general, a floating-point number is expressed as

$$N = M \cdot 2^E, \tag{18}$$

where M and E, both expressed in binary form, are termed the *mantissa* and the *exponent* of the number, respectively. In a binary floating-point representation, numbers are always normalized by scaling M to be a fraction whose

[3] When normalized signed numbers are added and the result is larger than one, then overflow occurs. Overflow does not occur in multiplication because the product of two normalized numbers is always less than one.

decimal value lies in the range $0.5 \leq M < 1$. When storing a floating-point number in a register, the register is divided into the mantissa and the exponent. Both the mantissa and the exponent have fictitious binary points to separate the sign bits from the numbers. In floating-point arithmetic, negative mantissas and negative exponents are coded the same way as in fixed-point arithmetic using signed-magnitude, two's complement, or one's complement representations. The product of two floating-point numbers is given by

$$(M_1 \cdot 2^{E_1})(M_2 \cdot 2^{E_2}) = (M_1 \times M_2) \cdot 2^{(E_1+E_2)}.$$

Thus, if the mantissa is limited to C bits, the product $M_1 \times M_2$ must be rounded or truncated to C bits. The sum of two floating-point numbers is performed by shifting the bits of the mantissa of the smaller number to the right and increasing its exponent until the two exponents are equal. The two mantissas are then added and if necessary normalized to satisfy Equation (18). It is possible that the shifted mantissa may exceed its limited range and thus must be quantized. Hence, in floating-point arithmetic, quantization errors are introduced in both addition and multiplication; therefore, roundoff or truncation errors will be introduced in the mantissa M but not in the exponent E. The reason is that the exponent, E, is always a positive or negative integer, and integers have exact binary representations. Of course, if the number is too large or too small, then overflow or underflow can occur.

3.2 Truncation and roundoff

Because of the finite word length of registers in digital computers, errors are always introduced when the numbers to be processed are quantized. These errors depend on (a) the way the numbers are represented (fixed- or floating-point arithmetic, signed-magnitude, two's or one's complement) and (b) how the numbers are quantized. Consider the normalized binary number $b_0 \bullet b_1 b_2 \cdots b_C$ where b_0 is the sign bit and $b_1 b_2 \cdots b_C$ is the binary code of a fixed-point number or the mantissa of a floating-point number. Denoting the number before quantization by x, the error introduced by quantization is given by

$$e_q = Q[x] - x,$$

where $Q[x]$ is the quantized value of x. The range of quantization error depends on the type of arithmetic and the type of quantization used. For fixed-point arithmetic, it can be shown [9] that the error caused by truncating a number to C bits is given by

$$\begin{aligned} -2^{-C} < e_T \leq 0, \quad & x \geq 0 \\ 0 \leq e_T < 2^{-C}, \quad & x < 0 \end{aligned} \tag{19}$$

for the signed-magnitude and one's complement representations. For a two's complement representation, the truncation error is given by

$$-2^{-C} < e_T \leq 0 \tag{20}$$

for all x. On the other hand, the error caused by rounding a number to C bits is given by

$$\frac{-2^{-C}}{2} \leq e_R < \frac{2^{-C}}{2} \tag{21}$$

for signed-magnitude, one's complement, and two's complement representations. In fixed-point arithmetic, truncation or roundoff errors are independent of the magnitude of the original unquantized numbers.

In floating-point arithmetic, roundoff and truncation errors depend on the magnitude of the unquantized number and occur only in the mantissa. Thus, if the mantissa is truncated to C bits, the quantized number is

$$x_q = (1 + e)x,$$

where e is the relative error in x. In the case of truncation, it can be shown that for signed-magnitude and one's complement representations, the relative error in the value of the floating-point word is

$$-2.2^{-C} < e \leq 0, \tag{22}$$

and for two's complement truncation, the error is

$$\begin{aligned} -2.2^{-C} < e \leq 0, \qquad & x \geq 0 \\ 0 \leq e < 2.2^{-C}, \; & x < 0. \end{aligned} \tag{23}$$

On the other hand, the roundoff error in the mantissa is of the form

$$-2^{-\mathrm{C}} \leq e \leq 2^{-\mathrm{C}} \tag{24}$$

for all three types of representations. The three major sources of error caused by finite word length are: (a) coefficient quantization, (b) quantization errors in A/D converters, and (c) quantization errors in arithmetic operations. These three error sources are now discussed along with their effects on the behavior of digital controllers.

3.3 Coefficient quantization

Usually, digital control system design methods result in controllers whose coefficients have infinite precision; however, because the controllers are implemented with finite word length registers, each of their coefficients must be quantized. For example, consider the digital controller described by the transfer function

$$H(z) = \frac{z^3 + 1.584z^2 + 1.2769z + 0.5642}{z^4 + 2.689z^3 + 3.3774z^2 + 2.3823z + 0.6942}, \tag{25}$$

which has poles located at $z_1 = 0.999, z_2 = 0.697, z_{3,4} = 0.4965 \pm j0.8663$ and is stable. If the binary forms of the coefficients of this controller are truncated to three bits to the right of the binary point, then the quantized controller transfer function becomes

$$H_q(z) = \frac{z^3 + 1.5z^2 + 1.25z + 0.5}{z^4 + 2.625z^3 + 3.3749z^2 + 2.3748z + 0.6249}, \tag{26}$$

which has two poles outside the unit circle and is, therefore, unstable. Another example is quoted in reference [10] in which a stable fifth-order controller can become unstable even if it is realized with 18-bit arithmetic.

In general, consider the digital controller described by the transfer function

$$H(z) = \frac{\sum_{k=0}^{m} b_k z^{-k}}{1 - \sum_{k=1}^{n} a_k z^{-k}}. \tag{27}$$

If the controller coefficients are quantized to C bits, then the quantized coefficients are

$$\hat{a}_k = a_k + \delta_k$$

for fixed-point arithmetic or

$$\hat{a}_k = a_k (1 + \delta_k)$$

for floating-point arithmetic where the quantization error δ_k is bounded in absolute value by 2^{-C}. Similarly,

$$\hat{b}_k = b_k + \eta_k$$

for fixed-point arithmetic or

$$\hat{b}_k = b_k (1 + \eta_k)$$

for floating-point arithmetic. In terms of the quantized coefficients, the controller transfer function becomes

$$H_q(z) = \frac{\sum_{k=0}^{m} \hat{b}_k z^{-k}}{1 - \sum_{k=1}^{n} \hat{a}_k z^{-k}}. \tag{28}$$

The most direct approach for analyzing the effects of coefficient quantization on system performance is referred to as *coefficient sensitivity analysis*. In this approach, the response of the quantized controller is compared with that of the ideal controller before quantization. The differences in the responses are called *variations*. For high order systems this is difficult to accomplish.

To simplify the coefficient sensitivity problem, higher order controller transfer functions can be decomposed, by factoring the numerator and the denominator of the transfer function, into cascade first- or second-order transfer functions. When all the poles and zeros of the controller transfer function are real, the cascaded transfer functions can all be of first order. Complex conjugate pairs of poles and zeros are grouped into second-order subsystems to avoid complex number arithmetic operations. Another way to simplify the coefficient sensitivity problem is to use a *parallel* form of the controller transfer function. The parallel form is obtained by decomposing the controller transfer function into first- or second-order subsystems using the method of partial fraction expansions.

Using either the cascade or the parallel form of the controller transfer function, each first-order subsystem has a transfer function of the form

$$H(z) = \frac{A(1 + \beta_1 z^{-1})}{1 - \alpha_1 z^{-1}} \tag{29}$$

and each second-order subsystem has a transfer function of the form

$$H(z) = \frac{A(1 + \beta_1 z^{-1} + \beta_2 z^{-2})}{1 - \alpha_1 z^{-1} - \alpha_2 z^{-2}}. \tag{30}$$

As a numerical example, consider again the controller transfer function given by (25). Rewriting the transfer function in factored form yields

$$H(z) = \left[\frac{z + 0.862}{z + 0.999}\right]\left[\frac{1}{z + 0.697}\right]\left[\frac{z^2 + 0.722z + 0.6545}{z^2 + 0.993z + 0.997}\right].$$

As in the previous example, if the binary representations of the coefficients of each factor are truncated to three bits to the right of the binary point, then the resulting quantized transfer function is

$$H_q(z) = \left[\frac{z + 0.75}{z + 0.875}\right]\left[\frac{1}{z + 0.625}\right]\left[\frac{z^2 + 0.625z + 0.625}{z^2 + 0.875z + 0.875}\right],$$

which is stable and can be realized by cascading first- and second-order subsystems of the form given in (29) and (30). Note that this controller is significantly different than the quantized controller given by (26).

3.4 Quantization in A/D conversion

The second source of error to be discussed is quantization in A/D conversion. A/D conversion involves two steps: sampling and quantization. In the quantization step, each sample of the sequence which has infinite precision is replaced by a digital code word of finite precision that is rounded or truncated to fit into a finite-length register. Rounding approximates the sample by the nearest *quantization level* and truncation approximates the sample by

the highest quantization level that is smaller than or equal to the sample value. Let the word length of the A/D converter be C bits and let the number converted be of the form

$$x_q = b_1 2^{-1} + b_2 2^{-2} + \cdots + b_C 2^{-C},$$

where $b_1, b_2, \cdots, b_C$ is the binary code. The sign bit, however coded, will always be present. The largest x_q that can possibly be produced by the A/D converter is

$$\begin{aligned} x_q &= 2^{-1} + 2^{-2} + 2^{-3} + \cdots + 2^{-C} = \tfrac{1}{2} \textstyle\sum_{i=0}^{C-1} \left(\tfrac{1}{2}\right)^i \\ &= 1 - 2^{-C} \end{aligned}$$

and the smallest non-zero x_q is 2^{-C} . The *dynamic range* of the A/D converter is defined to be the ratio of the largest value to the smallest value of x_q, that is,

$$DR = (1 - 2^{-C})/2^{-C} = 2^C - 1.$$

This implies that, to satisfy a requirement for a given DR , the number of bits must satisfy

$$C \geq \log_2 (DR + 1). \tag{31}$$

For roundoff, the error e is assumed to be a random variable with a probability density function, $f(x)$, that is uniformly distributed between $-q/2$ and $q/2$. The expected value of e is

$$E[e] = \int_{-\infty}^{\infty} x f(x) dx = \int_{-q/2}^{q/2} x dx = 0 \tag{32}$$

and the variance of e is

$$\text{var}(e) = E[e^2] = \int_{-\infty}^{\infty} x^2 f(x) dx = \int_{-q/2}^{q/2} (x^2/q) dx = q^2/12.$$

In terms of the number of bits, C, the variance is

$$\text{var}(e) = 2^{-2C}/12. \tag{33}$$

For signed-magnitude and one's complement truncation, the error is uniformly distributed between $-q$ and q; therefore, in these two cases, the expected value of e is

$$E[e] = \int_{-\infty}^{\infty} x f(x) dx = \int_{-q}^{q} (x/2q) dx = 0 \tag{34}$$

and the variance of e

$$\text{var}(e) = E[e^2] = \int_{-\infty}^{\infty} x^2 f(x)dx = \int_{-q}^{q} (x^2/2q)dx = q^2/3.$$

In terms of the number of bits, C,

$$\text{var}(e) = 2^{-2C}/3. \tag{35}$$

Comparing (35) with (33) for roundoff, quantization gives

$$\text{var}(e) = q^2/12 = 2^{-2C}/12 = 2^{-2(C+1)}/3.$$

For two's complement truncation, the error is uniformly distributed between -2^{-C} and zero; therefore,

$$E[e] = -2^{-C}/2 \tag{36}$$

and

$$\text{var}(e) = 2^{-2C}/12 = 2^{-2(C+1)}/3. \tag{37}$$

As discussed previously, the quantization error $e(k)$ can be viewed as an additive stationary white noise process with mean and variance given by (32), (33), (34), (35) or (36), (37) depending on whether the quantization is due to roundoff or truncation and what type of arithmetic is used. Using superposition and assuming that $x(k)$ and $e(k)$ are uncorrelated, the output of a digital controller can be decomposed into two parts, one due to the input $x(k)$ alone and the other due to $e(k)$ alone.

3.5 SNR of an A/D converter

If the input sequence $x(k)$ is modeled as a zero-mean Gaussian random sequence such that $3\sigma = 1$ (i.e., unity input level is a 3σ event), then the variance of the signal is

$$\sigma^2 = (1/3)^2 = 1/9.$$

Defining the *SNR* of an A/D converter as

$$\begin{aligned}(SNR)_{dB} &= 10 \cdot \log_{10}(\text{variance(unquantized signal)}/\text{variance(quantization error)}) \\ &= 10 \cdot \log_{10}(\text{var}(x(k))/\text{var}(e(k)))\end{aligned}$$

then for roundoff noise, the SNR is

$$\begin{aligned}(SNR)_{dB} &= 10 \cdot \log_{10}\left((1/9)/(2^{-2(C+1)}/3)\right) \\ &= 10 \cdot \log_{10}(1/3) - 10 \cdot \log_{10} 2^{-2(C+1)} = -4.77 + 6.02(C+1)\end{aligned} \tag{38}$$

or the minimum number of bits necessary for a given SNR is given by

$$C \geq 0.166(S/N)_{dB} - 0.207$$

For truncation noise, $(C+1)$ is replaced by C in (38). Hence,

$$(SNR)_{dB} = -4.77 + 6.02C \tag{39}$$

or the minimum number of bits for a given SNR is given by

$$C \geq 0.166(SNR)_{dB} + 0.792$$

(38) and (39) show that the SNR increases 6 dB for every additional bit added to the register. As a simple design procedure, one may choose the number of bits of the A/D converter to be greater than or equal to the larger of the two values necessary for (a) the required dynamic range and (b) the required SNR. That is,

$$C \geq \max\{\log_{10}(DR+1), 0.166(SNR)_{dB} - 0.207\} \tag{40}$$

for roundoff noise, or

$$C \geq \max\{\log_{10}(DR+1), 0.166(SNR)_{dB} + 0.792\} \tag{41}$$

for truncation noise.

The quantization error of an A/D converter need not be a serious problem. For example, in a 16-bit A/D converter, the maximum quantization error is $2^{-16} = 0.0015\%$ which is quite low compared to typical errors in analog sensors. This error, if taken to be "noise," gives an SNR of $20 \cdot \log_{10}(2^{-16}) = 96.3$ dB, which is much better than that of most high fidelity audio systems. Thus, the designer must simply ensure that enough bits are used to give the desired system accuracy.

3.6 Stochastic analysis of quantization errors in digital processing

One approach to analyzing roundoff and truncation errors that are generated by the arithmetic operations in digital controllers is to derive deterministic upper bounds on the maximum errors that can possibly result from roundoff or truncation [4]. In general, however, these bounds are *pessimistic* because the errors usually add up in the worst possible way. Another approach, used here, for analyzing these errors is to assume that they are stochastic noise sequences, develop stochastic models for these sequences, and then determine their effects on system performance using stochastic systems methods. In the remainder of this chapter, unless otherwise stated, the fixed- and floating-point errors are modeled as stationary white noise random sequences with probability density functions *uniformly distributed* over the range of quantization.

3.7 Fixed-point arithmetic

It was mentioned earlier that, in fixed-point arithmetic, quantization errors occur in multiplication and not in addition. Consider a multiplier in which two

C-bit numbers are multiplied and let $x(k)$ denote the product. The product $x(k)$ has length $2C$ and is quantized to form a number of length C which is denoted by $\hat{x}(k)$. The output of this multiplication/quantization operation can be written as the sum of the product $x(k)$ and the quantization error $e(k)$ (the difference between $x(k)$ and $\hat{x}(k)$). The quantization error is modeled as a stationary additive white noise sequence such that

$$E[e(k)] = \mu_e$$

$$E[e(i)e(j)] = \begin{cases} 0 & i \neq j \\ \text{var}(e) + \mu_e^2 & i = j \end{cases}$$

where μ_e and $\text{var}(e)$ are determined from appropriate choices taken from (32)–(37). If the product from the multiplier/quantizer is the input to a system, then using superposition, the output, $y_e(k)$, of a system due to the error $e(k)$ alone is given by the convolution sum

$$y_e(k) = \sum_{m=0}^{k} g(m)e(k-m),$$

where $g(m)$ is the unit pulse response of the system and the error source, $e(k)$, is the input, and $y_e(k)$ is the system output generated by the error. Assuming that the system is stable, it can easily be shown that as k approaches infinity, the mean and the variance of the output are given by

$$E\left[y_e(k)\right] = \mu_e \sum_{m=0}^{\infty} g(m) \tag{42}$$

and

$$\text{var}\left(y_e(k)\right) = \text{var}(e) \sum_{m=0}^{\infty} g^2(m), \tag{43}$$

respectively. Hence, the variance of the output equals the variance of the quantization noise times the *noise power gain*, NPG, where

$$NPG = \sum_{m=0}^{\infty} g^2(m). \tag{44}$$

Since digital controller transfer functions are usually realized using first- and second-order subsystems in parallel or cascade forms, the noise power gains of first- and second-order subsystems are now determined. The transfer function of a first-order subsystem of the form in (29) can be rewritten as

$$G(z) = k_0 + \frac{k_1}{1 - \alpha_1 z^{-1}}.$$

Hence

$$g(0) = k_0 + k_1$$
$$g(m) = k_1\alpha_1^m, \quad m = 1, 2, 3, \ldots$$

and

$$\sum_{m=0}^{\infty} g(m) = k_0 + \frac{k_1}{1-\alpha_1}.$$

Therefore, the mean of the output is

$$E[y_e(k)] = \mu_e \left(k_0 + \frac{k_1}{1-\alpha_1} \right). \tag{45}$$

Similarly,

$$g^2(0) = (k_0 + k_1)^2$$
$$g^2(m) = k_1^2\alpha_1^{2m}, \quad m = 1, 2, 3, \ldots$$

therefore, the noise power gain is

$$NPG = k_o^2 + 2k_0k_1 + \frac{k_1^2}{1-\alpha_1^2} \tag{46}$$

and

$$\mathrm{var}(y_e(k)) = \mathrm{var}(e) \left(k_o^2 + 2k_0k_1 + \frac{k_1^2}{1-\alpha_1^2} \right). \tag{47}$$

The transfer function of a second-order subsystem of the form in (30) can be written as

$$G(z) = k_o + \frac{k_1}{1-r_1z^{-1}} + \frac{k_2}{1-r_2z^{-1}},$$

therefore, the unit pulse response is

$$g(0) = k_0 + k_1 + k_2$$
$$g(m) = k_1r_1^m + k_2r_2^m, \quad m = 1, 2, 3, \ldots$$

and

$$\sum_{m=0}^{\infty} g(m) = k_o + \frac{k_1}{1-r_1} + \frac{k_2}{1-r_2} \tag{48}$$

and

$$E[y_e(k)] = \mu_e \left(k_o + \frac{k_1}{1-r_1} + \frac{k_2}{1-r_2} \right). \tag{49}$$

Similarly,

$$g^2(0) = (k_0 + k_1 + k_2)^2$$
$$g^2(m) = \left(k_1^2r_1^{2m} + 2k_1k_2r_1^mr_2^m + k_2^2r_2^{2m} \right), \quad m = 1, 2, \ldots$$

and the noise power gain is

$$NPG = k_o^2 + 2k_ok_1 + 2k_ok_2 + \frac{k_1^2}{1-r_1^2} + \frac{2k_1k_2}{1-r_1r_2} + \frac{k_2^2}{1-r_2^2}, \tag{50}$$

which is always a real number. The variance of the output due to the noise is

$$\mathrm{var}(y_e(k)) = \mathrm{var}(e)\left(k_o^2 + 2k_ok_1 + 2k_ok_2 + \frac{k_1^2}{1-r_1^2} + \frac{2k_1k_2}{1-r_1r_2} + \frac{k_2^2}{1-r_2^2}\right). \tag{51}$$

3.8 Quantization noise model of first-order subsystems

Consider a first-order subsystem preceded by an A/D converter as shown in Fig. 1(a). All of the quantization noise sources including the A/D conversion noise and the multiplication noises are in the model. Setting all the noise sources equal to zero, the transfer function of the ideal subsystem is

$$T(z) = \frac{Y(z)}{X(z)} = \frac{A(1+\beta_1 z^{-1}}{1-\alpha_1 z^{-1}}.$$

The effect of the A/D converter on the output can be determined by setting all the signals but e_0 to zero. The transfer function $G_0(z)$ relates $e_0(k)$ to the filter output $y_0(k)$ as

$$Y_0(z) = G_0(z)E_0(z),$$

where

$$G_0(z) = T(z) = k_0 + \frac{k_1}{1-\alpha_1 z^{-1}}.$$

Therefore, the mean of the output is given as, using (45),

$$E[y_0] = \mu_0\left(k_0 + \frac{k_1}{1-\alpha_1}\right)$$

and the variance of the output is, using (47),

$$\begin{aligned}\mathrm{var}(y_0) &= (NPG)_0\mathrm{var}(e_0)\\ &= \left(k_o^2 + 2k_0k_1 + \tfrac{k_1^2}{1-\alpha_1^2}\right)\mathrm{var}(e_0).\end{aligned}$$

The effect of quantization error in multiplication on the system output is determined as follows. All signals except e_a are set to zero, then

$$Y_a(z) = G_a(z)E_a(z) = T(z)E_a(z)$$

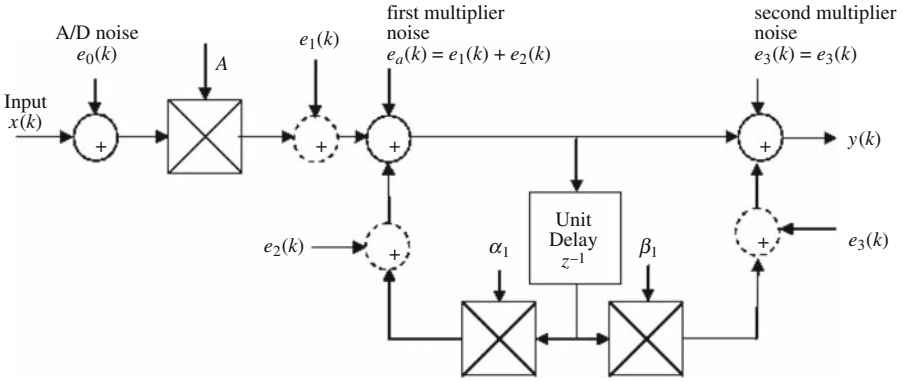
and

$$E[y_a(k)] = \mu_a\left(k_0 + \frac{k_1}{1-\alpha_1}\right)$$

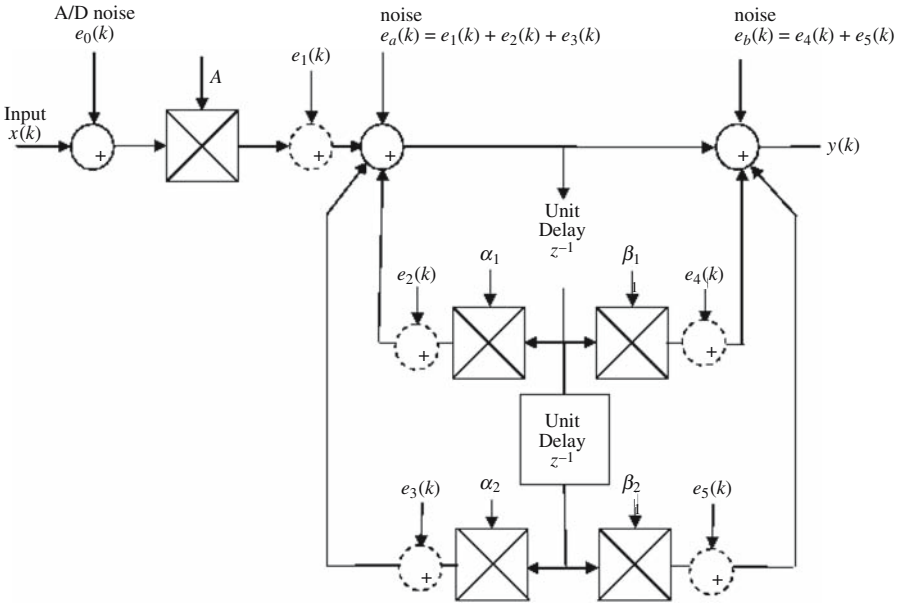
and

$$\mathrm{var}[y_a] = \left(k_0^2 + 2k_0k_1 + \frac{k_1^2}{1-\alpha_1^2}\right)\mathrm{var}(e_a).$$

Assuming the multiplier errors e_1 and e_2 are uncorrelated, then



(a) Fixed-point quantization noise model of first-order subsystem



(b) Fixed-point quantization noise model of second-order subsystem

Fig. 1. Models of quantization noise

$$\text{var}(e_1 + e_2) = \text{var}(e_1) + \text{var}(e_2).$$

Similarly, the second multiplier noise gain is calculated by setting all sources but e_3 to zero. The transfer function which relates e_3 to the output is

$$Y_3(z) = G_3(z)E_3(z),$$

where

$$G_3(z) = 1.$$

Then

$$E[y_3] = E[e_3]$$

and

$$\text{var}(y_3) = \text{var}(e_3).$$

Also, note that

$$(NPG)_3 = 1.$$

Assuming that the output noises y_0, y_a, and y_3 are uncorrelated, then

$$\begin{aligned}&\text{var}(y_0 + y_a + y_3)\\ &\quad = (NPG)_0\text{var}(e_0) + (NPG)_a(\text{var}(e_1) + \text{var}(e_2)) + (NPG)_3\text{var}(e_3).\end{aligned}$$

3.9 Quantization noise model of second-order subsystems

The previous analysis for first-order subsystems can be easily extended to second-order subsystems. The quantization noise model for second-order subsystems is shown in Fig. 1(b). Setting all the noise sources to zero, the ideal transfer function of the second-order subsystem, which relates $X(z)$ to $Y(z)$, is

$$T(z) = \frac{Y(z)}{X(z)} = \frac{A(1 + \beta_1 z^{-1} + \beta_2 z^{-2})}{1 - \alpha_1 z^{-1} - \alpha_2 z^{-2}},$$

which is that given in (30); therefore (49) and (51) can be used to determine statistics of the output noise. Assuming the output noises y_0, y_a, and y_b are uncorrelated, then

$$\begin{aligned}\text{var}(y_0 + y_a + y_b) = (NPG)_0\text{var}(e_0) + (NPG)_a(\text{var}(e_1) + \text{var}(e_2) + \text{var}(e_3))\\ +(NPG)_b(\text{var}(e_4) + \text{var}(e_5)),\end{aligned}$$

where

$$(NPG)_0 = (NPG)_a$$

is determined from $T(z)$ and

$$(NPG)_b = 1.$$

3.10 Floating-point arithmetic

The analysis of quantization errors in digital controllers that are implemented in floating-point arithmetic is more complicated than for those implemented in fixed-point arithmetic [11]. It was mentioned earlier that in floating-point arithmetic, quantization errors occur only in the mantissa and, therefore, roundoff and truncation errors are introduced in both addition and multiplication. For example, let x_1 and x_2 be any two numbers before quantization. Quantizing the sum and the product of these two numbers gives

$$(x_1 + x_2)_q = (x_1 + x_2)(1 + e_s) \tag{52}$$

$$(x_1 \cdot x_2)_q = (x_1 \cdot x_2)(1 + e_p), \tag{53}$$

respectively, where the relative errors e_s and e_p satisfy, depending on the number representation, (22)–(24). Each arithmetic operation introduces quantization errors according to (52) and (53). Detailed examples of roundoff and truncation errors accumulated in first- and second-order subsystems using floating-point arithmetic are given in [12].

3.11 Limit cycle and deadband effects

When digital controllers are implemented with finite word length, *limit cycles* (sustained oscillations) may appear at the controller output even in the absence of any applied input. Basically, there are two different kinds of limit cycles. One is due to roundoff in multiplication, termed the *deadband effect*, and the other is due to register overflow. Limit cycles exist in fixed-point digital controllers but can be ignored in floating-point controllers [11].

To illustrate the phenomenon of a limit cycle due to roundoff, consider the first-order controller described by the difference equation

$$y(k) = ay(k-1) + x(k), \tag{54}$$

where

$$x(k) = 0.9\delta(k) \quad a = 0.5, \quad y(-1) = 0.$$

If the controller equation is implemented with infinite word length registers, then

$$y(k) = 0.9(0.5)^k.$$

Note that as k approaches infinity the steady state value of the output, $y(k)$, approaches zero. However, assuming that the controller equation is implemented with a word length of 3 bits, then

$$y_q(k) = Q[0.5y_q(k-1)] + 0.75\delta(k).$$

Using a decimal representation, the output can be calculated recursively as follows:

$$
\begin{aligned}
y_q(0) &= Q[(0.5)(0)] + 0.75 = 0.75 \\
y_q(1) &= Q[(0.5)(0.75)] = 0.375 \\
y_q(2) &= Q[(0.5)(0.375)] = 0.25 \\
y_q(3) &= Q[(0.5)(0.25)] = 0.125 \\
y_q(4) &= Q[(0.5)(0.125)] = 0.125 \\
&\vdots \\
y_q(k) &= Q[(0.5)(0.125)] = 0.125 \\
&\vdots
\end{aligned}
$$

Hence, as k approaches infinity the steady state value of $y_q(k)$ approaches 0.125 and not zero.

As another example, again consider the system described by (54) and let

$$x(k) = 0, \quad a = -0.5, \quad y(-1) = 0.75.$$

If the controller equation is implemented with infinite word length registers, then the output

$$y(k) = 0.75(-0.5)^k$$

approaches zero as k approaches infinity. Assuming that the controller equation is implemented with a 3-bit word length, then

$$y_q(k) = Q[-0.5y_q(k-1)].$$

The output can be calculated recursively as follows:

$$
\begin{aligned}
y_q(0) &= Q[(-0.5)(0.75)] = Q[-0.375] = -0.375 \\
y_q(1) &= Q[(-0.5)(-0.375)] = 0.25 \\
y_q(2) &= Q[(-0.5)(0.25)] = -0.125 \\
&\vdots \\
y_q(k) &= Q[(-0.5)(0.125)] = -0.125 \\
&\vdots
\end{aligned}
$$

and oscillates between 0.125 and -0.125 indefinitely.

An interesting example of limit cycle due to register overflow is given in [13]. Limit cycles due to roundoff and overflow are usually undesirable and should not be permitted in a control system.

3.12 Effect of sampling rate on quantization error

In some applications, quantization errors cannot be ignored and their effect on the system output depends on the sampling period. As an example, consider an analog controller described by

$$H(s) = \frac{10^4}{s+1}. \tag{55}$$

Its discrete-time equivalent, using an impulse invariant approximation, is

$$H(z) = \frac{10^4 z}{z - e^{-T}} = \frac{10^4}{1 - e^{-T} z^{-1}} = k_0 + \frac{k_1}{1 - \alpha_1 z^{-1}},$$

where

$$k_0 = 0, \quad k_1 = 10^4, \quad \alpha_1 = e^{-T}$$

Assuming an A/D converter that uses roundoff and an infinite precision controller, the variance of the controller output noise, $y_\epsilon(k)$, equals the variance of the roundoff noise times the noise power gain of the controller. For a first-order controller, that variance is determined using (33) and (47), resulting in the variance of the output noise being a function of both the word length C and the sampling time T and given by

$$\mathrm{var}(y_\epsilon(k)) = \left(k_0^2 + 2k_0k_1 + \frac{k_1}{1-\alpha_1^2}\right)\frac{2^{-2C}}{12} = \frac{10^4 \cdot 2^{-2C}}{12(1 - e^{-2T})}.$$

Apparently, if C is fixed as the sampling period is decreased, the variance of the output noise is increased. Note that when $T = 0$, the discrete-time equivalent is unstable. Note also that, if T is fixed, increasing the word length C decreases the variance of the output noise. Other methods of forming discrete-time equivalents of the analog controller can lead to better results than those obtained in this example.

It is through computer simulation of the plant and the controller that the best sample rates are achieved. It is good practice to investigate carefully the behavior of the controlled system for various sample rates when the arithmetic precision of the controller is reduced, when disturbances and noises are injected into the system at likely points, and when the plant model is changed in ways that might occur in practice.

References

1. M.S. Santina, A.R. Stubberud, and G.H. Hostetter, *Digital Control System Design*, 2nd edition, International Thomson Publishing, Cincinnati, OH, 1994.
2. M.S. Santina and A.R. Stubberud, *Discrete-Time Equivalents to Continuous-Time Systems*, Eolss Publishers Co. Ltd., Oxford, UK, 2004.

3. K.J. Åström and B. Wittenmark, *Computer Controlled Systems*, 3rd edition, Prentice-Hall, Englewood Cliffs, NJ, 1996.
4. G.F. Franklin, J.D. Powell, and M.L. Workman, *Digital Control of Dynamic Systems*, 3rd edition, Addison-Wesley, Reading, MA, 1997.
5. M.S. Santina and A.R. Stubberud, *Sample-Rate Selection*, CRC Press, Boca Raton, FL, 1996.
6. M.S. Santina and A.R. Stubberud, *Quantization Effects*, CRC Press, Boca Raton, FL, 1996.
7. H. Hanselmann, Implementation of Digital Controllers-A Survey, *IFAC*, 23(1):7–32, 1987.
8. K. Hwang, *Computer Arithmetic*, Wiley, New York, 1979.
9. L.R. Rabiner, and B. Gold, *Theory and Application of Digital Signal Processing*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1975.
10. B. Gold, and C.M. Rader, *Digital Processing of Signals*, McGraw-Hill Inc., New York, 1969.
11. I.W. Sandberg, Floating-Point Round-off Accumulation in Digital Filter Realization, *Bell Syst. Tech. J.*, Vol. 46:1775–1791, 1967.
12. A.V. Oppenheim and R.W. Schafer, *Digital Signal Processing*, Prentice-Hall Inc., Englewood Cliffs, NJ, 1975.
13. P. Stubberud, *Digital Signal Processing*, Class Notes, University of Nevada, Las Vegas, NV, 1990.

Discrete-Event Systems

Christos G. Cassandras

Department of Manufacturing Engineering and
Center for Information and Systems Engineering,
Boston University, Brookline, MA 02446, U.S.A. cgc@bu.edu

1 Introduction

The term "discrete-event system (DES)" was introduced in the early 1980s to identify an increasingly important class of dynamic systems in terms of their most critical feature: the fact that their behavior is governed by *discrete events* occurring asynchronously over time and solely responsible for generating state transitions. In between event occurrences, the state of such systems is unaffected. Examples of such behavior abound in technological environments such as computer and communication networks, automated manufacturing systems, air traffic control systems, Command, Control, Communication, Computers, and Intelligence (C^4I) systems, advanced monitoring and control systems in automobiles or large buildings, intelligent transportation systems, distributed software systems, and so forth. The operation of such environments is largely regulated by human-made rules for initiating or terminating activities and scheduling the use of resources through controlled events, such as hitting a keyboard key, turning a piece of equipment "on,"or sending a message packet. In addition, there are numerous uncontrolled randomly occurring events, such as a spontaneous equipment failure or a packet loss, which may or may not be observable through sensors. We should point out that the acronym DEDS, for "discrete-event dynamic system,"is also commonly used to emphasize that it is the dynamics of such systems that render them particularly interesting [1],[2].

The conceptual and practical challenges in the study of DES may be summarized as follows:

1. The types of variables involved in the description of a DES are both continuous and discrete, sometimes purely symbolic, i.e., non-numeric (as in describing the state of a piece of equipment as "on" or "off"). This renders traditional mathematical models based on differential (or difference) equations inadequate, and methods relying on the power of calculus are, consequently, of limited use.

2. Because of the asynchronous nature of events that cause state transitions in DES, it is neither natural nor efficient to use time as a synchronizing element driving the system dynamics. It is for this reason that DES are often referred to as *event-driven*, to contrast them to classical *time-driven* systems based on the laws of physics; in the latter, as time evolves state variables such as position, velocity, temperature, pressure, current, voltage, etc., also continuously evolve. In order to capture event-driven state dynamics, however, different mathematical models are necessary.
3. Uncertainties are inherent in the technological environments where DES are encountered. Therefore, the mathematical models used for DES and all associated methods for analysis and control must incorporate this element of uncertainty, sometimes by explicitly modeling nondeterministic behavior and often through the inclusion of appropriate stochastic model components.
4. Complexity is also inherent in DES of practical interest, usually manifesting itself in the form of combinatorially explosive state spaces. Purely analytical methods for DES design, analysis, and control have proved to be limited. A large part of the progress made in this field has relied on the development of new paradigms characterized by a combination of mathematical techniques and effective processing of experimental data.

2 Event-Driven and Time-Driven Systems

There are two features that characterize DES. First, they usually involve at least some discrete quantities, typically measured by integer numbers (how many parts are in an inventory, how many planes are in a runway, how many telephone calls are active). Second, what drives these systems is a variety of instantaneous "events" such as the pushing of a button, hitting a keyboard key, or a traffic light turning green. In this section, we will explain how these features amount to fundamental differences between dynamic systems modeled through differential (or difference) equations and the class of DES.

Let us begin with the concept of "event." We do not attempt to formally define it, since it is a primitive concept with a good intuitive basis. We only wish to emphasize that an event should be thought of as occurring instantaneously and causing transitions from one system state value to another. An event may be identified with a specific action taken (e.g., pressing a button). It may be viewed as a spontaneous occurrence dictated by nature (e.g., a random computer failure) or it may be the result of several conditions which are suddenly all met (e.g., the fluid level in a tank exceeds a given value). For the purpose of developing a model for a DES, we will use the symbol e to denote an event. When considering a system affected by different types of events, we will assume that we can define an *event set* E whose elements are all these events. Clearly, E is a discrete set.

Let us next concentrate on the nature of the state space of a system. In continuous-state systems the state generally changes as time changes. This is particularly evident in discrete-time models: the "clock" is what drives a typical sample path. With every "clock tick" the state is expected to change, since *continuous* state variables *continuously* change with time. It is because of this property that we refer to such systems as *time-driven*. In this case, time is a natural independent variable which appears as the argument of all input, state, and output functions involved in modeling a system.

In DES, at least some of the state variables are discrete and their values change only at certain points in time through instantaneous transitions which we associate with "events." What is important is to specify the timing mechanism based on which events take place. Let us assume that there exists a clock through which we will measure time, and consider two possibilities: (*i*) At every clock tick an event e is selected from the event set E (if no event takes place, we can think of a "null event" as being a member of E, whose property is that it causes no state change), and (*ii*) At various time instants (not necessarily known in advance and not necessarily coinciding with clock ticks), some event e "announces" that it is occurring. There is a fundamental difference between (*i*) and (*ii*) above. In (*i*), state transitions are *synchronized* by the clock: there is a clock tick, an event (or no event) is selected, the state changes, and the process repeats. Thus, the clock alone is responsible for any possible state transition. In (*ii*), every event $e \in E$ defines a distinct process through which the time instants when e occurs are determined. State transitions are the result of combining these *asynchronous* concurrent event processes. Moreover, these processes need not be independent of each other. The distinction between (*i*) and (*ii*) gives rise to the terms *time-driven* and *event-driven* systems, respectively. Continuous-state systems are, by their nature, time-driven. However, in discrete-state systems this depends on whether state transitions are synchronized by a clock or occur asynchronously as in scheme (*ii*) above. Clearly, event-driven systems are more complicated to model and analyze, since there are several asynchronous event-timing mechanisms to be specified as part of our understanding of the system.

In view of this discussion, let us now turn our attention to mathematical models one can use for time-driven and event-driven systems. In the former case, the field of systems and control has based much of its success on the use of well-known differential-equation-based models, such as

$$\dot{\mathbf{x}}(t) = \mathbf{f}(\mathbf{x}(t), \mathbf{u}(t), t), \qquad \mathbf{x}(t_0) = \mathbf{x}_0 \tag{1}$$

$$\mathbf{y}(t) = \mathbf{g}(\mathbf{x}(t), \mathbf{u}(t), t), \tag{2}$$

where (1) is a (vector) state equation with initial conditions specified, and (2) is a (vector) output equation. As is common in system theory, $\mathbf{x}(t)$ denotes the state of the system, $\mathbf{y}(t)$ is the output, and $\mathbf{u}(t)$ represents the input, often associated with controllable variables used to manipulate the state so as to attain a desired output. In these models, it is normally assumed that

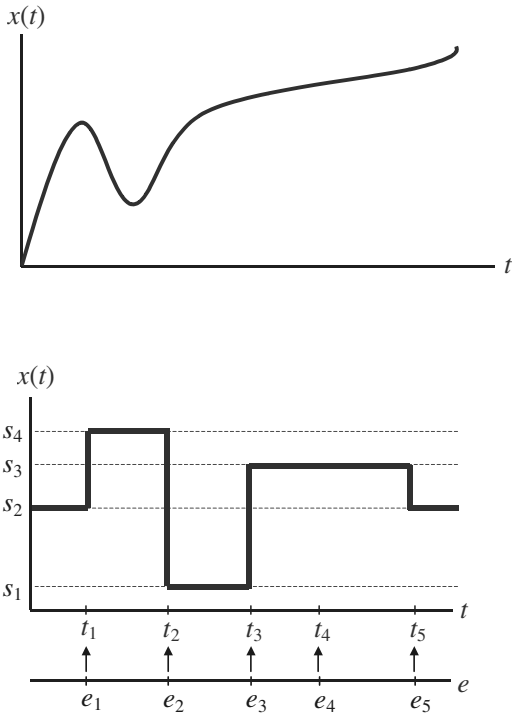


Fig. 1. Comparison of time-driven and event-driven sample paths

the state space is continuous and that the state transition mechanism is time-driven. Common physical quantities such as position, velocity, acceleration, temperature, pressure, flow, etc., fall in this category. Since we can naturally define time derivatives for these continuous variables, differential equation models like (1) can be used. The state generally changes as time changes and, as a result, the time variable t (or some integer $k = 0, 1, 2, \ldots$ in discrete time) is a natural independent variable for modeling such systems.

In contrast to a time-driven system, in a DES, where the state transition mechanism is event-driven, time no longer serves the purpose of driving such a system and may no longer be an appropriate independent variable. Comparing state trajectories (sample paths) of time-driven and event-driven systems is useful in understanding the differences between the two and setting the stage for DES modeling frameworks. Thus, comparing typical sample paths from each of these system classes, as in Fig. 1, we observe the following: (*i*) For the time-driven system shown, the state space X is the set of real numbers $\mathbb{R}$, and $x(t)$ can take any value from this set. The function $x(t)$ is the solution of a differential equation of the general form $\dot{x}(t) = f(x(t), u(t), t)$, where $u(t)$ is the input. (*ii*) For the event-driven system, the state space is some discrete set $X = \{s_1, s_2, s_3, s_4\}$. The sample path can only jump from one state to

another whenever an event occurs. Note that an event may take place, but not cause a state transition, as in the case of e_4. There is no immediately obvious analog to $\dot{x}(t) = f(x(t), u(t), t)$, i.e., no mechanism to specify how events might interact over time or how their time of occurrence might be determined. Thus, a large part of the early developments in the DES field has been devoted to the specification of an appropriate mathematical model containing the same expressive power as (1),(2) [2],[3],[4].

We should point out that discrete-*event* systems should not be confused with discrete-*time* systems. The class of discrete time systems contains both time-driven and event-driven systems.

3 Timed and Untimed Viewpoints of Discrete Event Systems

Let us return to the DES sample path shown in Fig. 1. Instead of plotting the piecewise constant function $x(t)$ as shown, it is often convenient to simply write the timed sequence of events

$$(e_1, t_1), (e_2, t_2), (e_3, t_3), (e_4, t_4), (e_5, t_5), \tag{3}$$

which contains the same information as the sample path depicted in Fig. 1. The first event is e_1 and it occurs at time $t = t_1$; the second event is e_2 and it occurs at time $t = t_2$, and so forth. When this notation is used, it is implicitly assumed that the initial state of the system, s_2 in this case, is known and that the system is "deterministic" in the sense that the next state after the occurrence of an event is unique. Thus, from the sequence of events in (3), we can recover the state of the system at any point in time and reconstruct the DES sample path in Fig. 1.

Consider the set of all possible timed sequences of events that a given system can ever execute. We call this set the *timed language* model of the system. The word "language" comes from the fact that we can think of the event E as an "alphabet" and of (finite) sequences of events as "words" [5]. We can further refine our model of the system if some statistical information is available about the set of sample paths of the system. Let us assume that probability distribution functions are available about the "lifetime" of each event type $e \in E$, that is, the elapsed time between successive occurrences of this particular e. We call a *stochastic timed language* a timed language together with associated probability distribution functions for the events. The stochastic timed language is then a model of the system that lists all possible sample paths together with relevant statistical information about them.

Stochastic timed language modeling is the most detailed in the sense that it contains event information in the form of event occurrences and their orderings, information about the exact times at which the events occur (and not only their relative ordering), and statistical information about successive

occurrences of events. If we omit the statistical information, then the corresponding timed language enumerates all the possible sample paths of the DES, with timing information. Finally, if we delete the timing information from a timed language we obtain an *untimed language*, or simply *language*, which is the set of all possible orderings of events that could happen in the given system. Deleting the timing information from a timed language means deleting the time of occurrence of each event in each timed sequence in the timed language. For example, the untimed sequence corresponding to the timed sequence of events in (3) is

$$\{e_1, e_2, e_3, e_4, e_5\}.$$

Untimed and timed languages represent different levels of abstraction at which DESs are modeled and studied. The choice of the appropriate level of abstraction clearly depends on the objectives of the analysis. In many instances, we are interested in the "logical behavior" of the system, that is, in ensuring that a precise *ordering of events* takes place which satisfies a given set of specifications (e.g., first-come first-served in a job processing system). Or we may be interested in finding if a particular state (or set of states) of the system can be reached or not. In this context, the actual timing of events is not required, and it is sufficient to model only the untimed behavior of the system, that is, to consider the language model of the system. *Supervisory control* is the term established for describing the systematic means (i.e., enabling or disabling events which are controllable) by which the logical behavior of a DES is regulated to achieve a given specification [6],[2].

Next, we may become interested in *event timing* in order to answer questions such as: "How much time does the system spend at a particular state?" or "How soon can a particular state be reached?" or "Can this sequence of events be completed by a particular deadline?" These and related questions are often crucial parts of the design specifications. More generally, event timing is important in assessing the performance of a DES often measured through quantities such as *throughput* or *response time*. In these instances, we need to consider the timed language model of the system. The fact that different event processes are concurrent and often interdependent in complex ways presents great challenges both for modeling and analysis of timed DESs. Moreover, since we cannot ignore the fact that DESs frequently operate in a stochastic setting (e.g., the time when some equipment fails is unpredictable), an additional level of complexity is introduced, necessitating the development of probabilistic models and related analytical methodologies for design and performance analysis based on stochastic timed language models. *Sample path analysis* refers to the study of sample paths of DESs, focusing on the extraction of information for the purpose of efficiently estimating performance sensitivities of the system and, ultimately, achieving on-line control and optimization [7],[2].

These different levels of abstraction are complementary, as they address different issues about the behavior of a DES. Indeed, the literature on DESs is

quite broad and varied as extensive research has been done on modeling, analysis, control, optimization, and simulation at all levels. Although the language-based approach to discrete event modeling is attractive in presenting modeling issues and discussing system-theoretic properties of DESs, it is by itself not convenient to address verification, controller synthesis, or performance issues. What is also needed is a convenient way of *representing* languages, timed languages, and stochastic timed languages. If a language (or timed language or stochastic timed language) is finite, we could always list all its elements, that is, all the possible sample paths that the system can execute. Unfortunately, this is rarely practical. Preferably, we would like to use *discrete event modeling formalisms* that would allow us to represent languages in a manner that highlights structural information about the system behavior and that is convenient to manipulate when addressing analysis and controller synthesis issues. In the next section, we provide a brief introduction to one of the major modeling formalisms, based on *automata*, which also forms the foundation for supervisory control and sample path analysis. We will use automata to illustrate the construction of models for a common class of DES and contrast it to two other modeling frameworks.

4 Modeling Overview

The introduction to DESs in the previous sections has served to point out the main characteristics of these systems. Two elements which have emerged as essential in defining a DES are (1) a discrete state space, which we denote by X, and (2) a discrete event set, which we denote by E. We can now build on this basic understanding in order to develop some formal models for DESs.

4.1 Automata

We already mentioned that the term "language" refers to the set of all possible event sequences that a given DES can execute. An *automaton* is a device that is capable of representing a language according to well-defined rules. We shall begin with the case of *deterministic* automata, and subsequently describe extensions to this concept.

Definition 1. A *deterministic automaton*, denoted by G, is a six-tuple

$$G = (X, E, f, \Gamma, x_0, X_m),$$

where X is the set of *states*, E is the finite set of *events* associated with the transitions in G, and $f : X \times E \to X$ is the *transition function*; specifically, $f(x, e) = y$ means that there is a transition labeled by event e from state x to state y and, in general, f is a *partial* function on its domain. $\Gamma : X \to 2^E$ is the *active event function* (or feasible event function); $\Gamma(x)$ is the set of all events e for which $f(x, e)$ is defined and it is called the *active event set* (or

feasible event set) of G at x. The notation 2^E means the *power set* of E, i.e., the set of all subsets of E. Finally, x_0 is the *initial* state and $X_m \subseteq X$ is the set of *marked states*.

The words *state machine* and *generator* (which explains the notation G) are also often used to describe the above object. Moreover, if X is a finite set (which we do not require in the definition above), we call G a *deterministic finite-state automaton*, often abbreviated as DFA. The automaton is said to be *deterministic* because f is a function over $X \times E$. In contrast, the transition structure of a *nondeterministic* automaton is defined by means of a relation over $X \times E \times X$ or, equivalently, a function from $X \times E$ to 2^X. Normally, the word "automaton" refers to a "deterministic automaton."

Regarding the inclusion of Γ in the definition of G, one of the reasons is so that we can distinguish between events e that are feasible at x but cause no state transition, that is, $f(x, e) = x$, and events e' that are not feasible at x, that is, $f(x, e')$ is undefined. Finally, regarding the set X_m, its selection depends on the problem of interest; for instance, this set is chosen to designate states which, when entered, indicate that the system has completed some operation or task.

The automaton G operates as follows. It starts in the initial state x_0 and upon the occurrence of an event $e \in \Gamma(x_0) \subseteq E$ it will make a transition to state $f(x_0, e) \in X$. This process then continues based on the transitions for which f is defined. Note that an event may occur without changing the state, i.e., it is possible that $f(x, e) = x$. It is also possible that two distinct events occur at a given state causing the exact same transition, i.e., for $a, b \in E$, $f(x, a) = f(x, b) = y$. What is interesting about the latter fact is that we may not be able to distinguish between events a and b by simply observing a transition from state x to state y.

For the sake of convenience, f is always extended from domain $X \times E$ to domain $X \times E^*$, where E^* is the set of *all* finite strings of elements of E, including the empty string (denoted by ε); the * operation is called the *Kleene closure*. This is accomplished in the following recursive manner: $f(x, \varepsilon) := x$ and $f(x, se) := f(f(x, s), e)$ for $s \in E^*$ and $e \in E$. The automaton model above is also referred to as a *generalized semi-Markov scheme* (abbreviated as GSMS) in the literature of stochastic processes. A GSMS is viewed as the basis for extending automata to incorporate an event timing structure and ultimately leads to *stochastic timed automata*, discussed next.

Let us start by considering an automaton G as defined above with a few minor changes: we allow for generally countable sets X and E, and we leave out of the definition any consideration for marked states. Thus, we begin with an automaton model (X, E, f, Γ, x_0). As it stands, this model is based on the premise that a given event sequence $\{e_1, e_2, \ldots\}$ is provided, so that, starting at state x_0, we can generate a state sequence $\{x_0, f(x_0, e_1), f(f(x_0, e_1), e_2), \ldots\}$. Note that if $\Gamma(x_0)$ includes several events, the fact that $e_1 \in \Gamma(x_0)$ is the particular event that occurs first is part of the input information necessary to operate the automaton. We extend our modeling setting to *timed* automata

by incorporating a *clock structure* associated with the event set E which now becomes the input from which a specific event sequence can be deduced.

Definition 2. The *clock structure* (or *timing structure*) associated with an event set E is a set $\mathbf{V} = \{\mathbf{v}_i : i \in E\}$ of clock (or lifetime) sequences

$$\mathbf{v}_i = \{v_{i,1}, v_{i,2}, \ldots\}, \quad i \in E, \quad v_{i,k} \in \mathbb{R}^+, \quad k = 1, 2, \ldots$$

We can then provide the following definition.

Definition 3. A *timed automaton* is a six-tuple

$$(X, E, f, \Gamma, x_0, \mathbf{V}),$$

where $\mathbf{V} = \{\mathbf{v}_i : i \in E\}$ is a clock structure and (X, E, f, Γ, x_0) is an automaton. The automaton generates a state sequence

$$x' = f(x, e') \tag{4}$$

driven by an event sequence $\{e_1, e_2, \ldots\}$ generated through

$$e' = \arg \max_{i \in \Gamma(x)} \{y_i\} \tag{5}$$

with the clock values y_i, $i \in E$, defined by

$$y_i' = \begin{cases} y_i - y^* \text{ if } i \neq e' \text{ and } i \in \Gamma(x) \\ v_{i,N_i+1} \text{ if } i = e' \text{ or } i \notin \Gamma(x) \end{cases} \qquad i \in \Gamma(x'), \tag{6}$$

where the *interevent time* y^* is defined as

$$y^* = \min_{i \in \Gamma(x)} \{y_i\} \tag{7}$$

and the *event scores* N_i, $i \in E$, are defined by

$$N_i' = \begin{cases} N_i + 1 \text{ if } i = e' \text{ and } i \notin \Gamma(x) \\ N_i \qquad \text{otherwise} \end{cases} \qquad i \in \Gamma(x'). \tag{8}$$

In addition, initial conditions are: $y_i = v_{i,1}$ and $N_i = 1$ for all $i \in \Gamma(x_0)$. If $i \notin \Gamma(x_0)$, then y_i is undefined and $N_i = 0$.

Comparing (4) to the state equation (1) for time-driven systems, we see that the former can be viewed as the event-driven analog of the latter. However, the simplicity of (4) is deceptive: unless an event sequence is given (as in the case of Definition 1), determining the *triggering* event e' which is required to obtain the next state x' involves the combination of (5)–(8). Therefore, the analog of (1) as a "canonical" state equation for a DES requires all the equations (5)–(8).

In Definition 3, the clock structure $\mathbf{V}$ is assumed to be fully specified in a deterministic sense. Let us now assume that the clock sequences $\mathbf{v}_i$, $i \in E$, are specified only as stochastic sequences. This means that we no

longer have at our disposal real numbers $\{v_{i,1}, v_{i,2}, \ldots\}$ for each event i, but rather a distribution function, denoted by F_i, which describes the *random* clock sequence $\{V_{i,k}\} = \{V_{i,1}, V_{i,2}, \ldots\}$.

Definition 4. The *stochastic clock structure* (or *stochastic timing structure*) associated with an event set E is a set of distribution functions $F = \{F_i : i \in E\}$ characterizing the stochastic clock sequences

$$\{V_{i,k}\} = \{V_{i,1}, V_{i,2}, \ldots\}, \quad i \in E, \quad V_{i,k} \in \mathbb{R}^+, \quad k = 1, 2, \ldots$$

Most of the DES analysis based on stochastic clock structures assumes that each clock sequence consists of random variables which are independent and identically distributed (iid) and that all clock sequences are mutually independent. Thus, each $\{V_{i,k}\}$ is completely characterized by a distribution function $F_i(t) = P[V_i \leq t]$. There are, however, several ways in which a clock structure can be extended to include situations where elements of a sequence $\{V_{i,k}\}$ are correlated or two clock sequences are dependent on each other.

We can extend the definition of a timed automaton by viewing the state, event, and all event scores and clock values as random variables denoted respectively by X, E, N_i, and Y_i, $i \in \mathcal{E}$, where we use $\mathcal{E}$ to denote the event set and distinguish it from some event E which takes values $i \in \mathcal{E}$. Similarly, we use $\mathcal{X}$ to denote the state space and distinguish it from some state X which takes values $x \in \mathcal{X}$.

Definition 5. A *stochastic timed automaton* is a six-tuple

$$(\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, F),$$

where $\mathcal{X}$ is a countable *state space*; $\mathcal{E}$ is a countable *event set*; $\Gamma(x)$ is the *active event set* (or feasible event set); $p(x';\ x, e')$ is a *state transition probability* defined for all $x, x' \in \mathcal{X}$, $e' \in \mathcal{E}$ and such that $p(x';\ x, e') = 0$ for all $e' \notin \Gamma(x)$; p_0 is the probability mass function $P[X_0 = x]$, $x \in \mathcal{X}$, of the initial state X_0; and F is a *stochastic clock structure*. The automaton generates a stochastic state sequence $\{X_0, X_1, \ldots\}$ through a transition mechanism (based on observations $X = x$, $E' = e'$):

$$X' = x' \text{ with probability } p(x';\ x, e') \tag{9}$$

and it is driven by a stochastic event sequence $\{E_1, E_2, \ldots\}$ generated through

$$E' = \arg \max_{i \in \Gamma(X)} \{Y_i\} \tag{10}$$

with the stochastic clock values Y_i, $i \in \mathcal{E}$, defined by

$$Y_i' = \begin{cases} Y_i - Y^* & \text{if } i \neq E' \text{ and } i \in \Gamma(X) \\ V_{i,N_i+1} & \text{if } i = E' \text{ or } i \notin \Gamma(X) \end{cases} \qquad i \in \Gamma(X'), \tag{11}$$

where the *interevent time* Y^* is defined as

$$Y^* = \min_{i \in \Gamma(X)} \{Y_i\} \tag{12}$$

and the *event scores* N_i, $i \in \mathcal{E}$, are defined by

$$N_i' = \begin{cases} N_i + 1 \text{ if } i = E' \text{ and } i \notin \Gamma(X) \\ N_i \qquad \text{otherwise} \end{cases} \qquad i \in \Gamma(X') \tag{13}$$

and $\{V_{i,k}\} \sim F_i$ (the notation $\sim$ denotes "with distribution"). In addition, initial conditions are: $X_0 \sim p_0(x)$, $Y_i = V_{i,1}$ and $N_i = 1$ if $i \in \Gamma(X_0)$. If $i \notin \Gamma(X_0)$, then Y_i is undefined and $N_i = 0$.

A simple interpretation of this elaborate definition is as follows. Given that the system is at some state X, the next event E' is the one with the smallest clock value among all feasible events $i \in \Gamma(X)$. The corresponding clock value, Y^*, is the interevent time between the occurrence of E and E', and it provides the amount by which the time, T, moves forward:

$$T' = T + Y^*.$$

Clock values for all events that remain active in state X' are decremented by Y^*, except for the triggering event E' and all newly activated events, which are assigned a new lifetime V_{i,N_i+1}. Event scores are incremented whenever a new lifetime is assigned to them. It is important to note that the "system clock" T is fully controlled by the occurrence of events, which cause it to move forward; if no event occurs, the system remains at the last state observed.

It is conceivable for two events to occur at the same time, in which case we need a priority scheme in order to overcome a possible ambiguity in the selection of the triggering event in (10). This is usually accomplished through a priority scheme over all events in $\mathcal{E}$. In practice, it is common to expect that every F_i in the clock structure is absolutely continuous over $[0, \infty)$ (so that its density function exists) and has a finite mean. This implies that two events can occur at exactly the same time only with probability 0.

A stochastic process $\{X(t)\}$ with state space $\mathcal{X}$ which is generated by a stochastic timed automaton $(\mathcal{X}, \mathcal{E}, \Gamma, p, p_0, F)$ is referred to as a *generalized semi-Markov process* (GSMP). This process is used as the basis of much of the sample path analysis methods for DESs (for details see [2],[8], [9]).

4.2 Petri nets

An alternative modeling formalism for the DES is provided by *Petri nets* , originating in the work of C. A. Petri in the early 1960s. Like an automaton, a Petri net is a device that manipulates events according to certain rules. One of its features is that it includes explicit conditions under which an event can be enabled. This representation is conveniently described graphically, at least for small systems, resulting in *Petri net graphs*. Petri net graphs are intuitive and capture a lot of structural information about the system. An automaton can

always be represented as a Petri net; on the other hand, not all Petri nets can be represented as *finite-state* automata. Another motivation for considering Petri net models of a DES is the body of analysis techniques that have been developed for studying them [10],[11].

The process of defining a Petri net involves two steps. First, we define the Petri net graph, also called the *Petri net structure*. Then we adjoin to this graph an initial state, a set of marked states, and a transition labeling function, resulting in the complete Petri net model, its associated dynamics, and the languages that it generates and marks.

Petri net graph. A Petri net is a *bipartite graph* with two types of nodes, *places* and *transitions*, and arcs connecting them. Events are associated with transition nodes. In order for a transition to occur, several conditions may have to be satisfied. Information related to these conditions is contained in place nodes. Some such places are viewed as the "input" to a transition; they are associated with the conditions required for this transition to occur. Other places are viewed as the output of a transition; they are associated with conditions that are affected by the occurrence of this transition. The precise definition of a Petri net graph is as follows.

Definition 6. A *Petri net graph* is a weighted bipartite graph (P, T, A, w) where P is the finite set of *places* (one type of node in the graph), T is the finite set of *transitions* (the other type of node in the graph), $A \subseteq (P \times T) \cup (T \times P)$ is the set of arcs from places to transitions and from transitions to places in the graph, and $w : A \to \{1, 2, 3, \ldots\}$ is the *weight function* on the arcs.

We assume that (P, T, A, w) has no isolated places or transitions. When drawing Petri net graphs, we need to differentiate between the two types of nodes, places and transitions. The convention is to use circles to represent places and bars to represent transitions. Let the set of places be represented by $P = \{p_1, p_2, \ldots, p_n\}$, and the set of transitions be represented by $T = \{t_1, t_2, \ldots, t_m\}$. A typical arc is of the form (p_i, t_j) or (t_j, p_i), and the weight related to an arc is a positive integer. In describing a Petri net graph, it is convenient to use $I(t_j)$ to represent the set of input places to transition t_j. Similarly, $O(t_j)$ represents the set of output places from transition t_j. Thus, we have

$$I(t_j) = \{p_i \in P : (p_i, t_j) \in A\}, \qquad O(t_j) = \{p_i \in P : (t_j, p_i) \in A\} \ .$$

Similar notation can be used to describe input and output transitions for a given place p_i: $I(p_i)$ and $O(p_i)$. An example of a Petri net graph for a simple DES is shown in Fig. 2 and discussed later in this section.

Petri net dynamics. *Tokens* are assigned to places in a Petri net graph in order to indicate the fact that the condition described by that place is satisfied. The way in which tokens are assigned to a Petri net graph defines a *marking*. Formally, a *marking* x of a Petri net graph (P, T, A, w) is a function $x : P \to \mathbb{N} = \{0, 1, 2, \ldots\}$. Marking x defines row vector $\mathbf{x} = [x(p_1), x(p_2), \ldots, x(p_n)]$, where n is the number of places in the Petri net. The ith entry of this vector indicates the (non-negative integer) number of tokens in place p_i, $x(p_i) \in \mathbb{N}$.

In Petri net graphs, a token is indicated by a dark dot positioned in the appropriate place.

The *state* of a Petri net is defined to be its marking row vector $\mathbf{x} = [x(p_1), x(p_2), \ldots, x(p_n)]$. Note that the number of tokens assigned to a place is an arbitrary non-negative integer, not necessarily bounded. It follows that the number of states we can have is, in general, infinite. Thus, the state space X of a Petri net with n places is defined by all n-dimensional vectors whose entries are non-negative integers, i.e., $X = \mathbb{N}^n$. While the term "marking" is more common than "state" in the Petri net literature, the term state is consistent with the role of state in system dynamics. Moreover, the term state avoids the potential confusion between marking in Petri net graphs and marking in the sense of *marked states* in automata. The state transition mechanism of a Petri net is captured by the structure of its graph and by "moving" tokens from one place to another. A transition $t_j \in T$ in a Petri net is said to be *enabled* if

$$x(p_i) \geq w(p_i, t_j) \quad \text{for all} \quad p_i \in I(t_j) \ .$$

In words, transition t_j in the Petri net is enabled when the number of tokens in p_i is at least as large as the weight of the arc connecting p_i to t_j, for all places p_i that are input to transition t_j.

When a transition is enabled, it can occur or *fire* (the term "firing" is standard in the Petri net literature). The *state transition function* of a Petri net is defined through the change in the state of the Petri net due to the firing of an enabled transition. The state transition function, $f : \mathbb{N}^n \times T \to \mathbb{N}^n$, of Petri net (P, T, A, w, x) is defined for transition $t_j \in T$ if and only if

$$x(p_i) \geq w(p_i, t_j) \quad \text{for all } p_i \in I(t_j) \ . \tag{14}$$

If $f(\mathbf{x}, t_j)$ is defined, then we set $\mathbf{x}' = f(\mathbf{x}, t_j)$ where

$$x'(p_i) = x(p_i) - w(p_i, t_j) + w(t_j, p_i), \quad i = 1, \ldots, n \ . \tag{15}$$

Condition (14) ensures that the state transition function is defined only for transitions that are enabled; an "enabled transition" is therefore equivalent to a "feasible event" in an automaton. But whereas in automata the state transition function was quite arbitrary, here the state transition function is based on the structure of the Petri net. Thus, the next state defined by (15) explicitly depends on the input and output places of a transition and on the weights of the arcs connecting these places to the transition. According to (15), if p_i is an input place of t_j, it loses as many tokens as the weight of the arc from p_i to t_j; if it is an output place of t_j, it gains as many tokens as the weight of the arc from t_j to p_i. Clearly, it is possible that p_i is both an input and output place of t_j, in which case (15) removes $w(p_i, t_j)$ tokens from p_i, and then immediately places $w(t_j, p_i)$ new tokens back in it.

In general, it is entirely possible that after several transition firings, the resulting state is $\mathbf{x} = [0, \ldots, 0]$, or that the number of tokens in one or more

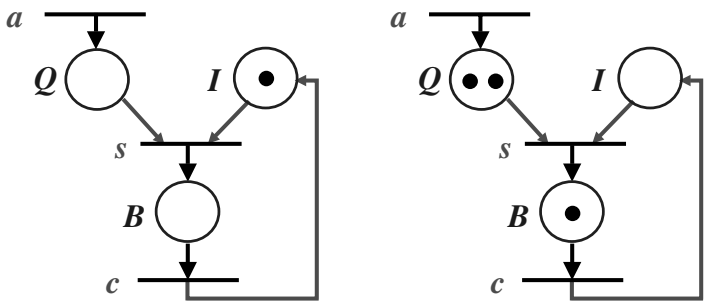


Fig. 2. A Petri Net example

places grows arbitrarily large after an arbitrarily large number of transition firings. The latter phenomenon is a key difference with automata, where finite-state automata have only a finite number of states, by definition. In contrast, a finite Petri net graph may result in a Petri net with an unbounded number of states.

In order to illustrate the use of Petri nets in modeling DESs, let us consider a simple queueing system, i.e., a system where "customers" indexed by $i = 1, 2, \ldots$ arrive at times a_i and are placed in a queueing area (the "queue") to await access to a "server" if that server is busy processing a prior customer at that time. We begin by considering three events (transitions) driving the system: a represents a customer arrival, s represents the start of customer service, and c represents service completion and departure from the system. Thus, we define the transition set $T = \{a, s, c\}$. Transition a is spontaneous and requires no conditions (input places). On the other hand, transition s depends on two conditions: the presence of customers in the queue, and the server being idle. We represent these conditions through two input places for this transition, place Q (queue) and place I (idle server). Finally, transition c requires that the server be busy, so we introduce an input place B (busy server) for it. Thus, our place set is $P = \{Q, I, B\}$. The complete Petri net graph for this system is shown in Fig. 2. On the left side, no tokens are placed in Q, indicating that the queue is empty, and a token is placed in I, indicating that the server is idle. This defines the initial state $\mathbf{x}_0 = [0, 1, 0]$. Since transition a is always enabled, we can generate several possible sample paths. As an example, on the right side of Fig. 2 we show state $[2, 0, 1]$ resulting from the transition firing sequence $\{a, s, a, a, c, s, a\}$. This state corresponds to two customers waiting in queue, while a third is in service (the first arrival in the sequence has already departed after transition c).

Similar to timed automata, we can define timed Petri nets by introducing a clock structure, except that now a clock sequence $\mathbf{v}_j$ is associated with a transition t_j. A positive real number, $v_{j,k}$, assigned to t_j has the following meaning: when transition t_j is enabled for the kth time, it does not fire immediately, but incurs a firing delay given by $v_{j,k}$; during this delay, tokens are

kept in the input places of t_j. Not all transitions are required to have firing delays. Some transitions may always fire as soon as they are enabled. Thus, we partition T into subsets T_0 and T_D, such that $T = T_0 \cup T_D$. T_0 is the set of transitions always incurring zero firing delay, and T_D is the set of transitions that generally incur some firing delay. The latter are called *timed transitions*.

Definition 7. The *clock structure* (or *timing structure*) associated with a set of timed transitions $T_D \subseteq T$ of a marked Petri net (P, T, A, w, x) is a set $\mathbf{V} = \{\mathbf{v}_j : t_j \in T_D\}$ of clock (or lifetime) sequences

$$\mathbf{v}_j = \{v_{j,1}, v_{j,2}, \ldots\},\ t_j \in T_D, \quad v_{j,k} \in \mathbb{R}^+, \quad k = 1, 2, \ldots$$

Graphically, transitions with no firing delay are still represented by bars, whereas timed transitions are represented by rectangles. The clock sequence associated with a timed transition is normally written next to the rectangle.

Definition 8. A *timed Petri net* is a six-tuple

$$(P, T, A, w, x, \mathbf{V}),$$

where (P, T, A, w, x) is a marked Petri net and $\mathbf{V} = \{\mathbf{v}_j : t_j \in T_D\}$ is a clock structure.

4.3 Dioid algebras

Another modeling framework we will briefly describe in what follows is based on developing an algebra using two operations: $\min\{a, b\}$ (or $\max\{a, b\}$) for any real numbers a and b, and addition $(a+b)$. The motivation comes from the observation that the operations "min" and "+" are the only ones required to develop the timed automaton model in Definition 3. Similarly, the operations "max" and "+" are the only ones used in developing the timed Petri net models described in the previous section. The term "dioid" (meaning "two") refers to the fact that this algebra is based on two operations . The operations are formally named *addition* and *multiplication* and denoted by $\oplus$ and $\otimes$ respectively. However, their actual meaning (in terms of regular algebra) is different. For any two real numbers a and b, we define

$$Addition: \qquad a \oplus b \equiv \max\{a, b\} \tag{16}$$

$$Multiplication: \qquad a \otimes b \equiv a + b. \tag{17}$$

This dioid algebra is also called a $(\max, +)$ algebra [12],[3]. The motivation for pursuing this modeling approach goes beyond the observation that "max" and "+" are the only operations of apparent importance in the study of DES. If we consider a standard linear discrete-time system, its state equation is of the form

$$\mathbf{x}(k+1) = \mathbf{A}\mathbf{x}(k) + \mathbf{B}\mathbf{u}(k),$$

which involves (regular) multiplication $(\times)$ and addition $(+)$. It turns out that we can use a $(\max, +)$ algebra with DES, replacing the $(+, \times)$ algebra of

conventional time-driven systems, and come up with a representation similar to the one above, thus paralleling to a considerable extent the analysis of classical time-driven linear systems.

We will illustrate the point above by considering a common class of DESs encountered in resource contention management where a queueing model describes the process of tasks waiting to gain access to a particular resource (e.g., packets waiting for transmission at a network switch). Such systems have an event set $\mathcal{E} = \{a, d\}$, where a denotes an "arrival" event, and d denotes a "departure" event. A natural state variable is the number of tasks in the queue, which we shall call the *queue length*. By convention, the queue length at time t is allowed to include a task in process at time t. Thus, the state space is the set of non-negative integers, $\mathcal{X} = \{0, 1, 2, \ldots\}$. Let us define a_k to be the kth arrival time and d_k to be the kth departure time. The clock structure for this model (see Definition 2) consists of $\mathbf{v}_a = \{v_{a,1}, v_{a,2}, \ldots\}$ and $\mathbf{v}_d = \{v_{d,1}, v_{d,2}, \ldots\}$. It can then be shown that for $k = 1, 2, \ldots$

$$a_k = a_{k-1} + v_{a,k}, \qquad a_0 = 0 \tag{18}$$

$$d_k = \max\{a_{k-1} + v_{a,k}, d_{k-1}\} + v_{d,k}, \qquad d_0 = 0. \tag{19}$$

For simplicity, we assume that $v_{a,k} = C_a$, $v_{d,k} = C_d$ for all $k = 1, 2, \ldots$, where C_a and C_d are given constants. Note that C_a represents a constant interarrival time, and C_d represents a constant processing time. We will also assume that $C_a > C_d$, which is reasonable to ensure stability of the queueing system. We now rewrite the equations above as follows:

$$\begin{aligned} a_{k+1} &= a_k + C_a, & a_1 &= C_a \\ d_k &= \max\{a_k, d_{k-1}\} + C_d, & d_0 &= 0. \end{aligned}$$

Using the $(\max, +)$ algebra, these relationships can also be expressed as

$$\begin{aligned} a_{k+1} &= (a_k \otimes C_a) \oplus (d_{k-1} \otimes -L) \\ d_k &= (a_k \otimes C_d) \oplus (d_{k-1} \otimes C_d), \end{aligned}$$

where $-L$ is any sufficiently small negative number so that $\max\{a_k + C_a, d_{k-1} - L\} = a_k + C_a$. In matrix notation, we have

$$\begin{bmatrix} a_{k+1} \\ d_k \end{bmatrix} = \begin{bmatrix} C_a & -L \\ C_d & C_d \end{bmatrix} \begin{bmatrix} a_k \\ d_{k-1} \end{bmatrix}. \tag{20}$$

Defining

$$\mathbf{x}_k = \begin{bmatrix} a_{k+1} \\ d_k \end{bmatrix}, \qquad \mathbf{A} = \begin{bmatrix} C_a & -L \\ C_d & C_d \end{bmatrix}$$

we get

$$\mathbf{x}_{k+1} = \mathbf{A}\mathbf{x}_k, \qquad \mathbf{x}_0 = \begin{bmatrix} C_a \\ 0 \end{bmatrix}, \tag{21}$$

which in fact looks like a standard linear system model, except that the addition and multiplication operations are in the $(\max, +)$ algebra. We should emphasize, however, that while conceptually this offers an attractive representation of the queueing system's event timing dynamics, from a computational standpoint one still has to confront the complexity of performing the "max" operation when numerical information is ultimately needed to analyze the system or to design controllers for its proper operation.

5 Control and Optimization of Discrete Event Systems

The various control and optimization methodologies developed to date for DESs depend on the modeling level appropriate for the problem of interest.

Logical behavior. Issues such as ordering events according to some specification or ensuring the reachability of a particular state are normally addressed through the use of automata (see Definition 1) and Petri nets (see Definition 6). Supervisory control theory provides a systematic framework for formulating and solving problems of this type. In its simplest form, a *supervisor* is responsible for enabling or disabling a subset of events that are controllable, so as to achieve a desired behavior. The supervisor's actions follow the occurrence of those events that are in fact observable (generally, a subset of the event set of the DES). In this manner, a supervisor may be viewed as a feedback controller. Similar to classical control theory, two central concepts here are those of *controllability* and *observability*. The synthesis of supervisors is a particularly challenging problem, largely due to the inherent computational complexity resulting from the combinatorial growth of typical state spaces of DESs. Some special classes of supervisor synthesis problems and solution approaches for them are discussed in [6],[13], and a comprehensive coverage of supervisory control can be found in [2].

Event timing. When timing issues are introduced, timed automata (see Definition 3) and timed Petri nets (see Definition 8) are invoked for modeling purposes. Supervisory control in this case becomes significantly more complicated. An important class of problems, however, does not involve the ordering of individual events, but rather the requirement that selected events occur within a given "time window" or with some desired periodic characteristics. Models based on the algebraic structure of timed Petri nets or the $(\max, +)$ algebra provide convenient settings for formulating and solving such problems [11],[3].

Performance analysis. As in classical control theory, one can define a performance (or cost) function intended to measure how "close" one can get to a desired behavior or simply as a convenient means for quantifying system behavior. This approach is particularly crucial in the study of stochastic DESs, where the design of a system is often based on meeting specifications defined through appropriate performance metrics, such as the expected response time of tasks in a computer system or the throughput of a manufacturing process.

Because of the complexity of DES dynamics, analytical expressions for such performance metrics in terms of controllable variables are seldom available. This has motivated the use of simulation and, more generally, the study of DES sample paths; these have proven to contain a surprising wealth of information for control purposes. The theory of *perturbation analysis* has provided a systematic way of estimating performance sensitivities with respect to system parameters for important classes of DES. What is noteworthy in this approach is that these sensitivity estimates can be extracted from a *single* sample path of a DES, resulting in a variety of efficient algorithms which can often be implemented on line. Perturbation analysis and related approaches are covered in [8],[9],[14],[2].

Simulation. Because of the aforementioned complexity of DES dynamics, simulation becomes an essential part of DES performance analysis [15]. Discrete event simulation can be defined as a systematic way of generating sample paths of a DES by means of the timed automaton in Definition 3 or its stochastic counterpart in Definition 5. In the latter case, a computer pseudo-random number generator is responsible for the task of providing the elements of the clock sequences in the model. It should also be clear that the same process can be carried out using a Petri net model or one based on the dioid algebra setting.

Optimization. Optimization problems can be formulated in the context of both untimed and timed models of DESs. Moreover, such problems can be formulated in both a deterministic and a stochastic setting. In the latter case, the ability to efficiently estimate performance sensitivities with respect to controllable system parameters provides a powerful tool for stochastic gradient-based optimization (when one can define derivatives) [16] or discrete optimization methods (in the frequent cases where the controllable parameters of interest are discrete, as in turning a piece of equipment "on" or "off" or selecting the (integer) number of resources needed to execute certain tasks).

Hybrid systems. It is worth mentioning that the combination of time-driven and event-driven dynamics gives rise to a *hybrid* system [17],[18]. Control and optimization methodologies for such systems are particularly challenging as they need to capture both aspects, often through appropriate decomposition approaches.

References

1. Y. C. Ho (Ed), *Discrete Event Dynamic Systems: Analyzing Complexity and Performance in the Modern World*. New York: IEEE Press, 1991.
2. C. G. Cassandras and S. Lafortune, *Introduction to Discrete Event Systems*. Dordrecht: Kluwer Academic Publishers, 1999.
3. F. Baccelli, G. Cohen, G. J. Olsder, and J. P. Quadrat, *Synchronization and Linearity*. New York: Wiley, 1992.
4. P. Glasserman and D. D. Yao, *Monotone Structure in Discrete-Event Systems*. New York: Wiley, 1994.

5. J. E. Hopcroft and J. Ullman, *Introduction to Automata Theory, Languages, and Computation*. Reading, MA: Addison-Wesley, 1979.
6. P. J. Ramadge and W. M. Wonham, Supervisory control of a class of discrete event processes, *SIAM Journal on Control and Optimization*, vol. 25, no. 1, pp. 206–230, 1987.
7. Y. C. Ho and C. G. Cassandras, A new approach to the analysis of discrete event dynamic systems, *Automatica*, vol. 19, pp. 149–167, 1983.
8. Y. C. Ho and X. Cao, *Perturbation Analysis of Discrete Event Dynamic Systems*. Dordrecht: Kluwer Academic Publishers, 1991.
9. P. Glasserman, *Gradient Estimation via Perturbation Analysis*. Dordrecht: Kluwer Academic Publishers, 1991.
10. J. L. Peterson, *Petri Net Theory and the Modeling of Systems*. Englewood Cliffs, NJ: Prentice-Hall, 1981.
11. J. O. Moody and P. Antsaklis, *Supervisory Control of Discrete Event Systems Using Petri Nets*. Boston, MA: Kluwer Academic Publishers, 1998.
12. R. A. Cuninghame-Green, Minimax algebra, in *Number 166 in Lecture Notes in Economics and Mathematical Systems*, Berlin: Springer-Verlag, 1979.
13. E. Chen and S. Lafortune, Dealing with blocking in supervisory control of discrete event systems, *IEEE Trans. on Automatic Control*, vol. AC-36, no. 6, pp. 724–735, 1991.
14. C. G. Cassandras and C. G. Panayiotou, Concurrent sample path analysis of discrete event systems, *Journal of Discrete Event Dynamic Systems: Theory and Applications*, vol. 9, pp. 171–195, 1999.
15. A. M. Law and W. D. Kelton, *Simulation Modeling and Analysis*. New York: McGraw-Hill, 1991.
16. F. J. Vázquez-Abad, C. G. Cassandras, and V. Julka, Centralized and decentralized asynchronous optimization of stochastic discrete event systems, *IEEE Trans. Automatic Control*, vol. 43, no. 5, pp. 631–655, 1998.
17. M. S. Branicky, V. S. Borkar, and S. K. Mitter, A unified framework for hybrid control: Model and optimal control theory, *IEEE Trans. on Automatic Control*, vol. 43, no. 1, pp. 31–45, 1998.
18. P. J. Antsaklis (Ed), *Proceedings of the IEEE, Special Issue on Hybrid Systems*, vol. 88., 2000.

Introduction to Hybrid Systems

Michael S. Branicky

Department of Electrical Engineering and Computer Science
Case Western Reserve University
Cleveland, OH 44106, U.S.A.
mb@case.edu

Summary. Hybrid systems arise when the continuous and the discrete meet. Combine continuous and discrete inputs, outputs, states, or dynamics, and you have a hybrid system. Particularly, hybrid systems arise from the use of finite-state logic to govern continuous physical processes (as in embedded control systems) or from topological and network constraints interacting with continuous control (as in networked control systems). This chapter provides an introduction to hybrid systems, building them up first from the completely continuous side and then from the completely discrete side. It should be accessible to control theorists and computer scientists alike.

1 Hybrid Systems All Around Us

Hybrid systems arise in embedded control when digital controllers, computers, and subsystems modeled as finite-state machines are coupled with controllers and plants modeled by partial or ordinary differential equations or difference equations. Thus, such systems arise whenever one mixes logical decision making with the generation of continuous-valued control laws. These systems are driven on our streets, used in our factories, and flown in our skies; see Fig. 1.

Adding to the complexity is the case where sensing, control, and actuation are not hardwired but connected by a shared network medium; see Fig. 2. Such *networked control systems* (*NCSs*) are an important class of hybrid control systems [40]. The hybrid nature inherent in embedded control is further complicated by (i) the asynchronous or *event-driven* nature of data transmission, due to sampling schemes, varying transmission delay, and packet loss; and (ii) the discrete implementation of the network and its protocols, involving packets of data, queuing, routing, scheduling, etc.

So, hybrid systems arise in embedded and networked control. More specifically, real-world examples of hybrid systems include systems with relays, switches, and hysteresis [33,37]; computer disk drives [18]; transmissions, stepper motors, and other motion controllers [14]; constrained robotic systems [4];

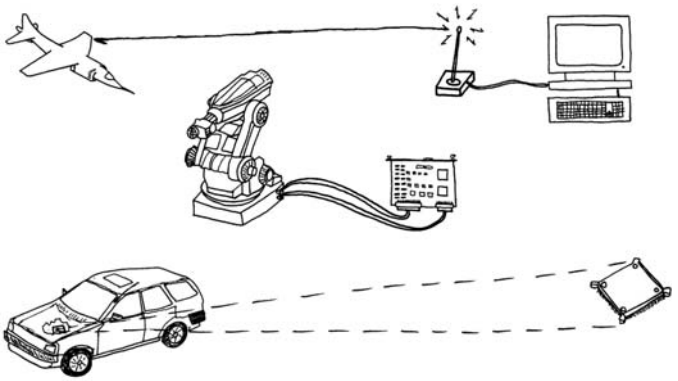


Fig. 1. Hybrid systems arising from embedded control

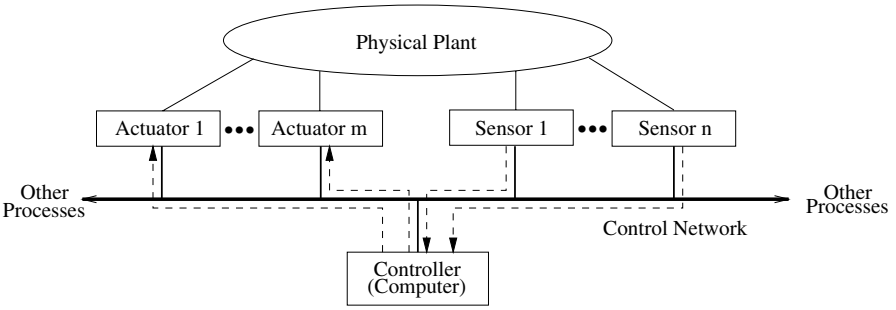


Fig. 2. Typical networked control system setup and information flows [40]

automated highway systems (AHSs) [36]; flight control and management systems [27, 35]; multi-vehicle formations and coordination [32]; analog/digital circuit codesign and verification [26]; and biological applications [17]. Next, we examine some of these in more detail.

Systems with switches and relays: Physical systems with switches and relays can naturally be modeled as hybrid systems. Sometimes, the dynamics may be considered merely discontinuous, such as in a blown fuse. In many cases of interest, however, the switching mechanism has some hysteresis, yielding a discrete state on which the dynamics depends. This situation is depicted by the multi-valued function H shown in Fig. 3(left). Suppose that the function H models the hysteretic behavior of a thermostat. Then a thermostatically controlled room may be modeled as follows:

$$\dot{x} = f(x, H(x - x_0), u), \tag{1}$$

where x and x_0 denote actual and desired room temperature, respectively. The function f denotes the dynamics of temperature, which depends on the current temperature, whether the furnace is switched On or Off, and some

auxiliary control signal u (e.g., the fuel burn rate). Note that this system is not just a differential equation whose right-hand side is piecewise continuous. There is "memory" in the system, which affects the value of the vector field. Indeed, such a system naturally has a finite automaton associated with the hysteresis function H, as pictured in Fig. 3(right). The notation ![condition] denotes that the transition *must* be taken when "enabled." That is, the event of x attaining a value greater than or equal to Δ triggers the discrete or *phase* transition of the underlying automaton from $+1$ to -1.

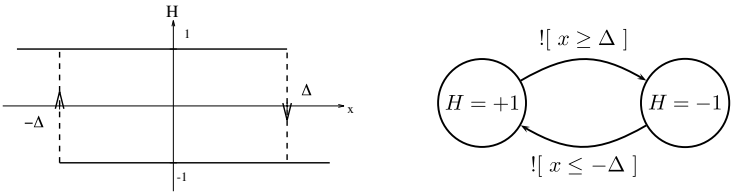


Fig. 3. (left) Hysteresis function, H, (right) finite automaton associated with H

Disk drive: Once a computer disk drive is up and spinning, it receives external commands to find data. The action of the disk drive is modeled by the differential (or difference equations) capturing the dynamic behavior of the disk, spindle, disk arm, and motors. The drive receives symbolic inputs of disk sectors and locations, it waits until the head is settled on the appropriate cylinder, begins a read operation and then transmits symbolic outputs corresponding to the bytes read (see Fig. 4). Again, the edge labels ![condition] denote transitions taken as soon as the condition is true; Read and Seek(Adr) are symbolic commands issued from another element/process in the system. While the logic governing this cycle of operation is simple, the vast majority of logic inside a disk drive (not shown, see [18]) is for startup, shutdown, and error handling.

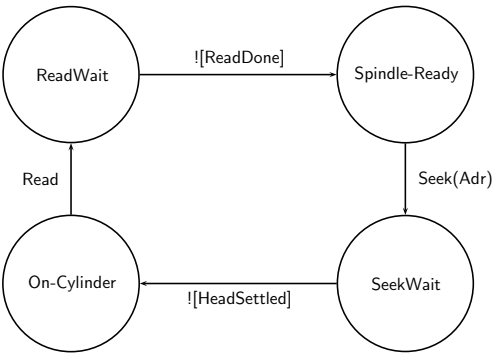


Fig. 4. Hybrid system associated with main disk drive functionality

Hopping robot control: Constrained robotic systems are interesting examples of hybrid systems. In particular, consider the hopping robots of Marc Raibert [29]. The dynamics of these devices are governed by gravity, as well as the forces generated by passive and active (pneumatic) springs. The dynamics change abruptly at certain event times and fall into distinct phases: Flight, Compression, Thrust, and Decompression; see Fig. 5(left). In fact, Raibert has built controllers for these machines that embed a finite-state machine that transitions according to these detected phases; see Fig. 5(right). There, the variable h is the distance of the hopper's base from the ground, and x is the displacement of the spring from equilibrium; T is an auxiliary timer, and the notation /action denotes that on transitioning, T is reset to zero; $\wedge$ denotes logical "and." Therefore, the transition from Flight to Compression occurs when touchdown is detected; that from Decompression to Flight upon liftoff. The switch from Compression to Thrust is controller initiated: when it detects the bottom-most point of compression, it activates the pneumatic cylinders to open for a fixed period of time, τ_{thrust}, after which Decompression of the pneumatic spring continues. Thus, finite automata and differential equations naturally interact in such devices and their controllers.

Vehicle powertrains: An automobile transmission system takes the continuous inputs of accelerator position and engine RPM and the discrete input of gear position and translates them into the motion of the vehicle. Likewise, the engine has discrete cylinders, which are fired at certain event times, as well as the continuous variables of fuel/air mixture and temperature. Finally, the whole powertrain is governed by a number of networked microprocessors to achieve some overall goal. For example, they may be coordinated by a cruise control system that accelerates and decelerates under different profiles. The desired profile is chosen depending on sensor readings (e.g., continuous reading of elevation, discrete coding of road condition, etc.). In such a case, we are to design a control system with both continuous and discrete inputs and outputs, whose internal components themselves are hybrid. This is typical in hybrid control; see Fig. 6. There, I and O are discrete (i.e., countable) sets of symbols and U and Y are continuums.

AHS: A more complicated example of a hybrid system arises in the control structures for an automated highway system (AHS) [36]. The basic goal of one such system is to increase highway throughput by means of a technique known as *platooning*. A platoon is a group of between, say, one and twenty vehicles traveling closely together in a highway lane at high speeds. To ensure safety—and proper formation and dissolution of structured platoons from the "free agents" of single vehicles—requires a bit of control effort! Protocols for basic maneuvers such as Merge, Split, and ChangeLane have been proposed in terms of finite-state machines. More conventional controllers govern the engines and brakes of individual vehicles. Clearly, the system is hybrid, with each vehicle

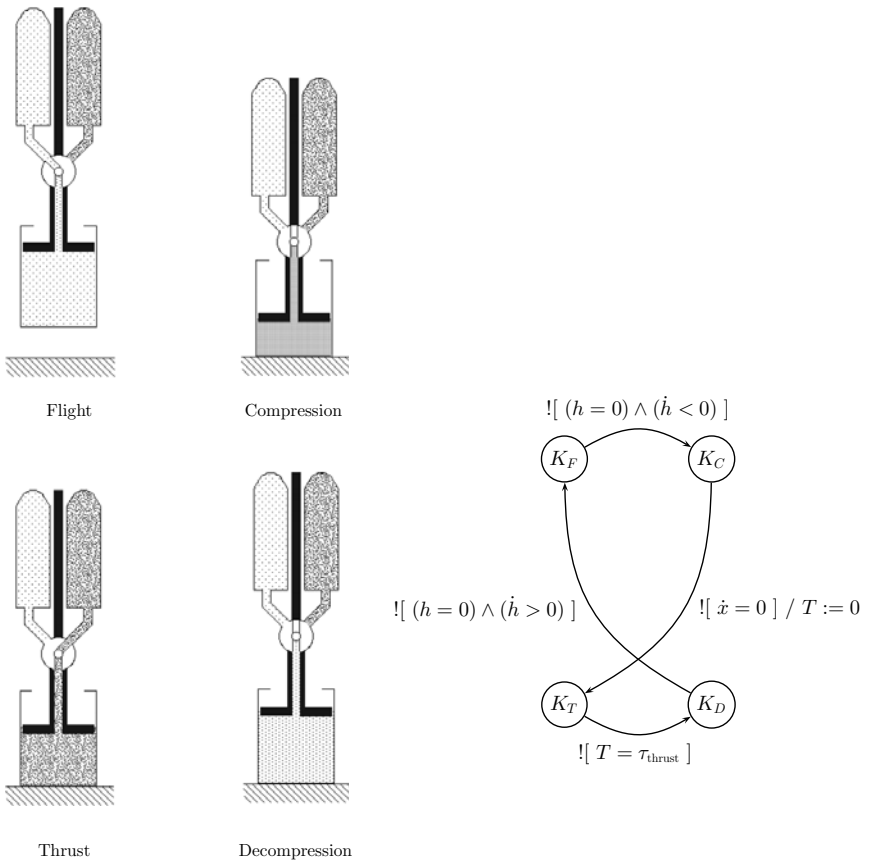


Fig. 5. Raibert's hopping robot: (left) dynamic phases (reproduced from [4], p. 260, Fig. 3, © Springer-Verlag), (right) finite-state controller with transitions specified

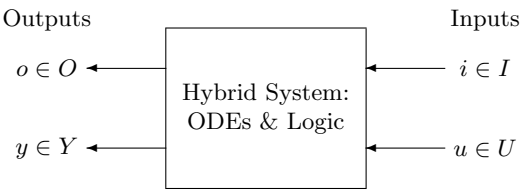


Fig. 6. Block diagram of prototypical hybrid control system

having a state determined by continuous variables (such as velocity, engine RPM, and distance to car ahead) and the finite states of its protocol-enacting state machines.

Flight control: Flight controllers are organized around the idea of *modes*. For example, one might easily imagine different control schemes for Take-Off, Ascend, Cruise, Descend, and Land. More complex is a whole flight vehicle management system, which coordinates flight in these different regimes, while also planning flight paths considering air traffic, weather, fuel economy, and passenger comfort [27].

Preview

Now that we have given a taste of the hybrid nature of real applications, we turn to developing mathematical models that can capture their behavior. Instead of proceeding directly to such hybrid models, we build them up, step by step, from well-known models of completely continuous (viz., ODEs in Section 2) and completely discrete (viz., finite automata in Section 3) systems. Then, in Section 4, we present an overarching model of hybrid systems that combines ODEs and automata. Throughout, we fix ideas using examples.

2 From Continuous Toward Hybrid

In this section, we review ordinary differential equations (ODEs) as a base continuous model,[1] then show how to add various discrete phenomena to them, as seen in the applications above.

2.1 Base continuous model: ODEs

In this chapter, the base continuous dynamical systems dealt with are defined by the solutions of *ODEs*:

$$\dot{x}(t) = f(x(t)), \tag{2}$$

where $x(t) \in X \subset \mathbf{R}^n$. The function $f : X \longrightarrow \mathbf{R}^n$ is called a *vector field* on $\mathbf{R}^n$. We assume existence and uniqueness of solutions.[2]

Actually, the system of ODEs in (2) is called *autonomous* or *time invariant* because its vector field does not depend explicitly on time. If it did depend

[1] One can easily use difference equations instead. See [7].

[2] See [22] for conditions. A well-known *sufficient* condition is that f is *Lipschitz continuous*. That is, there exists $L > 0$ (called the *Lipschitz constant*) such that

$$\|f(x) - f(y)\| \leq L\|x - y\|, \qquad \text{for all } x, y \in X.$$

explicitly on time, it would be *nonautonomous* or *time varying*, which one might explicitly note using the following notation:

$$\dot{x}(t) = f(x(t), t). \tag{3}$$

An *ODE with inputs and outputs* [25,31] is given by

$$\begin{aligned} \dot{x}(t) &= f(x(t), u(t)), \\ y(t) &= h(x(t), u(t)), \end{aligned} \tag{4}$$

where $x(t) \in X \subset \mathbf{R}^n$, $u(t) \in U \subset \mathbf{R}^m$, $y \in Y \subset \mathbf{R}^p$, $f : \mathbf{R}^n \times \mathbf{R}^m \longrightarrow \mathbf{R}^n$, and $h : \mathbf{R}^n \times \mathbf{R}^m \longrightarrow \mathbf{R}^p$. The functions $u(\cdot)$ and $y(\cdot)$ are the *inputs* and *outputs*, respectively. Whenever inputs are present, as in (4), we say that $f(\cdot)$ is a *controlled vector field*.

Differential inclusions: A *differential inclusion* allows the derivative to belong to a set and is written as

$$\dot{x}(t) \in F(x(t)),$$

where $F(x(t))$ is a set of vectors in $\mathbf{R}^n$. It can be used to model nondeterminism, including that arising from controls or disturbances. For example, the controlled differential equation of (4) can be viewed as an inclusion by setting $F(x) = \cup_{u \in U} f(x, u)$.

Example 1 (Innacurate Clock). A clock that has a time-varying rate between 0.9 and 1.1 can be modeled by $\dot{x} \in [0.9, 1.1]$. Such an inclusion is called a *rectangular* inclusion. ◇

2.2 Adding discrete phenomena

We have seen that hybrid systems are those that involve continuous states and dynamics, as well as some *discrete phenomena* corresponding to discrete states and dynamics. As described above, our focus in this chapter is on the case where the continuous dynamics is given by a differential equation

$$\dot{x}(t) = \xi(t), \qquad t \geq 0. \tag{5}$$

Here, then, $x(t)$ is considered the *continuous component* of the hybrid state, taking values in some subset $\mathbf{R}^n$. The vector field $\xi(t)$ generally depends on $x(t)$ and the aforementioned discrete phenomena.

Hybrid *control* systems are those that involve continuous states, dynamics, and controls, as well as discrete phenomena corresponding to discrete states, dynamics, and controls. Here, $\xi(t)$ in (5) is a *controlled* vector field which generally depends on $x(t)$, the *continuous component* $u(t)$ of the control policy, and the aforementioned discrete phenomena.

In this section, we identify the discrete phenomena alluded to above that generally arise in hybrid systems. They are as follows:

- autonomous switching, where the vector field changes discontinuously;
- autonomous jumps, where the state changes discontinuously;
- controlled switching, where a control switches vector fields discontinuously;
- controlled jumps, where a control changes a state discontinuously.

Next, we examine each of these discrete phenomena in turn, giving examples.

Autonomous switching

Autonomous switching is the phenomenon where the vector field $\xi(\cdot)$ changes discontinuously when the continuous state $x(\cdot)$ hits certain "boundaries" [3, 28, 33, 37]. The simplest example of this is when it changes depending on a "clock" which may be modeled as a supplementary state variable [14]. Another example of autonomous switching is the hysteresis function from Section 1.

Example 2 (Furnace). Consider the problem of controlling a household furnace. The temperature dynamics may be quite complicated, depending on outside temperature, humidity, luminosity; insulation and layout; whether incandescent lights are on, doors are closed, vents are open, people are present; and many other factors. Thus, let's just say that when the furnace is On, the dynamics are given by $\dot{x}(t) = f_1(x(t))$, where $x(t)$ is the temperature at time t; likewise, when the furnace is Off, let's say that the dynamics are given by $\dot{x}(t) = f_0(x(t))$. The full system dynamics are that of a *switched system*:

$$\dot{x}(t) = f_{q(t)}(x(t)),$$

where $q(t) = 0$ or 1 depending on whether the furnace is Off or On, respectively. ◇

A particular class of hybrid systems of interest is *switched linear systems*:

$$\dot{x}(t) = A_{q(t)}x(t), \qquad q \in \{1, \ldots, N\},$$

where $x(t) \in \mathbf{R}^n$ and each $A_q \in \mathbf{R}^{n\times n}$. Such a system would be autonomous if the switchings were a function of the state $x(t)$. Sometimes the "switching rules" might interact with the constituent dynamics to produce unexpected results, as in the next example.

Example 3 (Unstable from Stable). Consider A_1 and A_2 where

$$A_1 = \begin{bmatrix} -0.1 & 1 \\ -10 & -0.1 \end{bmatrix}, \qquad A_2 = \begin{bmatrix} -0.1 & 10 \\ -1 & -0.1 \end{bmatrix}.$$

Then $\dot{x} = A_i x$, is globally exponentially stable for $i = 1, 2$. But the switched system using A_1 in the second and fourth quadrants and A_2 in the first and third quadrants is unstable. Fig. 7 plots ten seconds of trajectories for each of A_1, A_2 and the switched system starting from $(1, 0)$, $(0, 1)$, $(10^{-6}, 10^{-6})$, respectively. In each plot, motion is clockwise. In the first two, the range shown is $[-3, 3] \times [-3, 3]$; in the last, it is $[-2 \cdot 10^5, 8 \cdot 10^5] \times [-5 \cdot 10^4, 3.5 \cdot 10^5]$. ◇

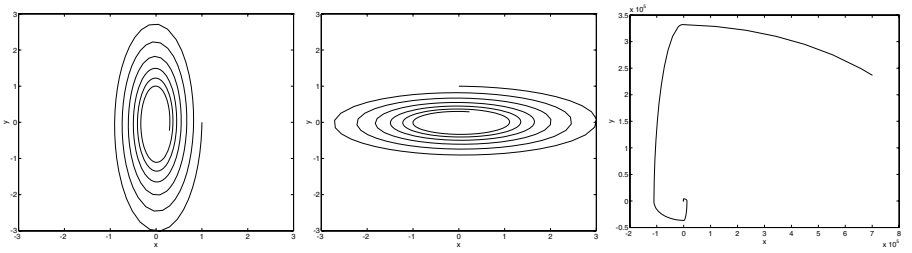


Fig. 7. Trajectories for (left) A_1x, (center) A_2x, and (right) a switched linear system

Autonomous jumps

An *autonomous jump* is the phenomenon where the continuous state $x(\cdot)$ jumps discontinuously on hitting prescribed regions of the state space [4,5]. We may also call these *autonomous impulses*. The simplest examples possessing this phenomenon are those involving collisions.

Example 4 (Bouncing Ball). Consider the case of the vertical motion of a ball of mass m under gravity with constant g. The dynamics are given by

$$\begin{aligned} \dot{x} &= v, \\ \dot{v} &= -mg. \end{aligned}$$

Further, upon hitting the ground (assuming downward velocity), we instantly set v to $-\rho v$, where $\rho \in [0, 1]$ is the coefficient of restitution. We can encode the jump in velocity as a rule by saying

If at time t, $x(t) = 0$ and $v(t) < 0$, then $v(t^+) = -\rho v(t)$.

In this case, $v(\cdot)$ is piecewise continuous (from the right), with discontinuities occurring when $x = 0$. This "rule" notation is quite general, but cumbersome. We have found it more desirable to use the following equational notation:

$$v^+(t) = -\rho v(t), \qquad (x(t), v(t)) \in \{(0, v) \mid v < 0\}$$

Here, we have used Sontag's evocative discrete-time transition notation [31] to denote the "successor" of $x(t)$. ◇

A general system subject to autonomous impulses may be written as

$$\begin{aligned} \dot{z}(t) &= f(z(t)), & z(t) &\notin A \\ z^+(t) &= G(z(t)), & z(t) &\in A. \end{aligned} \tag{6}$$

The interpretation of these equations is that the dynamics evolves according to the differential equation while z is in the complement of the *autonomous jump set* A, but that the state is immediately reset according to the map G upon z's hitting the set A.

Controlled switching

Controlled switching is the phenomenon where the vector field $\xi(\cdot)$ changes abruptly in response to a control command, usually with an associated cost. This can be interpreted as switching between different vector fields [38]. Controlled switching arises, for instance, when one is allowed to pick among a number of vector fields:

$$\dot{x} = f_q(x), \qquad q \in Q \simeq \{1, 2, \ldots, N\}.$$

Here, the q that is active at any given time is to be chosen by the controller. If one were to make the choice an explicit function of state, then the result would be a closed-loop system with autonomous switches.

Example 5 (Satellite Control). In satellite control, one encounters

$$\ddot{\theta} = \tau_{\text{eff}} v,$$

where θ, $\dot{\theta}$ are the angular position and velocity, respectively, and $v \in \{-1, 0, 1\}$, depending on whether the reaction jets are full reverse, off, or full on. ◇

Example 6 (Transmission). This example includes controlled switching and continuous controls. Consider a simplified model of a manual transmission, modified from one in [14]:

$$\begin{aligned} \dot{x}_1 &= x_2, \\ \dot{x}_2 &= [-a(x_2/v) + u]/(1 + v), \end{aligned}$$

where x_1 is the ground speed, x_2 is the engine RPM, $u \in [0, 1]$ is the throttle position, and $v \in \{1, 2, 3, 4\}$ is the gear shift position. The function a is positive for a positive argument. ◇

Controlled jumps

A *controlled jump* is the phenomenon where the continuous state $x(\cdot)$ changes discontinuously in response to a control command, usually with an associated cost [6]. We also call these jumps *controlled impulses*.

Example 7 (Inventory Management). In a simple inventory management model [6], there are a "discrete" set of restocking times $\theta_1 < \theta_2 < \cdots$ and associated order amounts $\alpha_1, \alpha_2, \ldots$. The equations governing the stock at any given moment are

$$\dot{x}(t) = -\mu(t) + \sum_i \delta(t - \theta_i)\alpha_i,$$

where μ represents degradation/utilization and δ is the Dirac delta. Note: If one makes the stocking times and amounts an explicit function of x (or t), then these controlled jumps become autonomous jumps. ◇

Example 8 (Planetary Flybys). Exploration spacecraft typically use close encounters with moons and planets to gain energy and change course. At the level of the entire solar system, these maneuvers are planned by considering the flight path to be a sequence of parabolic curves, with resets of heading and velocity occurring at the "point" of encounter. ◇

3 From Discrete to Hybrid

In this section, we review finite-state machines, or finite automata, as a base discrete model. We then successively add timing and continuous dynamics, arriving at models that approach more general hybrid systems.

3.1 Base discrete model: Automata

The base discrete model is usually a finite-state machine, finite transition system, or finite automaton, for which there is a very rich and beautiful theory [23]. These are actually discrete dynamical systems that process inputs. Therefore, the direct correspondent to the autonomous ODE of (2) would be an *inputless automaton*, which is a dynamical system with a discrete state space:

$$q_{k+1} = \nu(q_k),$$

where $q_k \in Q$, a finite set.

Example 9 (Finite Counter). A finite counter (modulo N) is an inputless automaton with $Q = \{0, 1, \ldots, N-1\}$ with $\nu(q) = q + 1$. ◇

Preliminaries: Before defining automata with input, we begin with some standard material for discussing discrete inputs. A *symbol* is the abstract entity of automata theory. Examples are letters and digits. An *alphabet* is a finite set of symbols. For example, the English alphabet is $A = \{\mathtt{a}, \mathtt{b}, \mathtt{c}, \ldots, \mathtt{z}\}$ and the binary alphabet is $B = \{\mathtt{0}, \mathtt{1}\}$. A *string* or a *word* (over alphabet I) is a finite sequence of juxtaposed symbols from I. Thus, `cat` and `jazz` and `zebra` are strings over A. So are `w` and `qqq`. The *empty string*, denoted ε, is the string consisting of zero symbols. The set of all strings over an alphabet I is denoted by I^*. The set of all binary strings is

$$B^* \equiv \{\varepsilon, \mathtt{0}, \mathtt{1}, \mathtt{00}, \mathtt{01}, \mathtt{10}, \mathtt{11}, \mathtt{000}, \mathtt{001}, \ldots\}.$$

A *language* (over alphabet I) is merely a set of strings over I. Thus, the words in a dictionary form a language over A, namely, the English language, $E \subset A^*$. Obviously, $\mathtt{cat} \in E$ and $\mathtt{qqq} \notin E$. The set of all binary strings of length two is a language over B, as is the set of all binary strings of even length. So is B^*. Yet another is the strings of odd parity (those having an odd number of ones):

$$B_{odd} \equiv \{\mathtt{1}, \mathtt{01}, \mathtt{10}, \mathtt{001}, \ldots\}.$$

The empty set is the *empty language*, namely the language consisting of no strings whatsoever. Note that it is different than the language $\{\varepsilon\}$.

DFA: We are now ready to define a *(deterministic) finite automaton* (*DFA*) which is a four-tuple $\mathcal{A} = (Q, I, \nu, q_0)$, where

- Q is a finite set of *states*,
- I is an alphabet, called the *input alphabet*,
- ν is the *transition function* mapping $Q \times I$ into Q, and
- $q_0 \in Q$ is the *initial state*.

The idea is that the machine above begins in state q_0 and then responds to or processes input strings over I. In one move, a DFA in state q receives/processes symbol $a \in I$ and enters state $\nu(q, a)$. On input word $w = a_1 a_2 \cdots a_n$, the DFA in state r_0 successively processes symbols and sequences through states $r_1, r_2, \ldots, r_n$, such that

$$r_{k+1} = \nu(r_k, a_{k+1}); \qquad k = 0, 1, 2, \ldots, n-1. \tag{7}$$

This sequence is called a *run* of the DFA over w.

Example 10 (Parity). Consider DFA $\mathcal{A}_{\text{par}} = (\{q_0, q_1\}, \{\mathtt{0}, \mathtt{1}\}, \nu, q_0)$, with

$$\nu(q_0, \mathtt{1}) = q_1; \qquad \nu(q_1, \mathtt{1}) = q_0; \qquad \nu(q_i, \mathtt{0}) = q_i, \ i \in \{1, 2\};$$

see Fig. 8(left). It keeps track of the parity of its input string by counting 1s, modulo 2. On input 111, it has run q_0, q_1, q_0, q_1; on ε, it has run q_0. ◇

NFA: The base definition of a DFA can be easily augmented in various ways. For example, a *nondeterministic finite automaton* (*NFA*) would allow a *set* of start states and a set-valued transition function mapping $Q \times I$ into 2^Q, so that at any stage the automaton may be in a set of states consistent with its transition function and the symbols seen so far. In one move, an NFA in state q receives/processes symbol a and nondeterministically enters any one of the states in $\nu(q, a)$. On input word $w = a_1 a_2 \cdots a_n$, an NFA in state r_0 nondeterministically sequences through states $r_1, r_2, \ldots, r_n$ such that

$$r_{k+1} \in \nu(r_k, a_{k+1}); \qquad k = 0, 1, 2, \ldots, n-1.$$

Again, such a sequence is a *run* of the NFA over w. In general, an NFA has many runs associated with each string; a DFA only has one.

Marked states: Actually, the traditional definitions of DFA and NFA add a set of marked or accepting or *final states*, F. In computation theory, one then defines the *language of A*, which is the set of strings *accepted* by the machine: the set of strings that have at least one run that ends in a state in F. For example, if $F = \{q_1\}$ in Example 10 above, 111 is accepted and ε is not. In fact, the language of $\mathcal{A}_{\text{par}}$ in this case is B_{odd}. If $F = \{q_0, q_1\}$, the accepted language would be B^*; if $F = \{q_1\}$, it would be $B^* - B_{\text{odd}}$; if $F = \emptyset$, it is $\emptyset$.

ω-Automata: Yet another way to augment the definition above is to allow machines that process infinite sequences of symbols. An *ω-string* (read "omega string") or *ω-word* over an alphabet I is an infinite-length sequence of symbols from I. For example, the following are ω-strings over B:

$$\mathtt{0}^\omega \equiv \mathtt{0000}\cdots \qquad \mathtt{1}^\omega \equiv \mathtt{1111}\cdots \qquad (\mathtt{01})^\omega \equiv \mathtt{0101}\cdots$$

Likewise, ω-languages are sets of ω-words. *ω-automata* act in exactly the same way as finite automata, and can be deterministic or not. Namely, on input ω-word $w = a_1 a_2 \cdots$, a DFA in state r_0 would successively process symbols and sequence through states $r_1, r_2, \ldots$, satisfying (7). Again, this sequence of states is a *run* over w. In Example 10, the run over 1^ω is $q_0, q_1, q_0, q_1, \ldots$.

More general automata: Finally, one can add discrete outputs and allow a countable (versus finite) number of states, resulting in a digital or symbolic or discrete automaton, with input and output. A *discrete automaton* is a machine (Q, I, ν, O, η) consisting of the state space, input alphabet, transition function, output alphabet, and output function, respectively. We assume that Q, I, and O are each countable. When these sets are finite, the result is a finite automaton with output. In any case, the functions involved are $\nu : Q \times I \longrightarrow Q$ and $\eta : Q \times I \longrightarrow O$. The *dynamics* of the automaton are specified by

$$\begin{aligned} q_{k+1} &= \nu(q_k, i_k), \\ o_k &= \eta(q_k, i_k). \end{aligned} \tag{8}$$

Such a model is easily seen to encompass finite automata and ω-automata, as well as other models from the literature (Mealy and Moore machines, push-down automata, Turing machines, Petri nets, etc.) [15, 23].

Example 11 (Mealy Machine). Consider the discrete automaton (derived from [23]; see Fig. 8) $\mathcal{A}_{\text{same}} = (\{q_\varepsilon, q_0, q_1\}, \{0,1\}, \nu, \{\mathtt{n}, \mathtt{y}\}, \eta)$, with

$$\nu(q_\varepsilon, i) = q_i,\ \nu(q_i, 0) = q_0,\ \nu(q_i, i) = q_1;\ \ \eta(q_\varepsilon, i) = \mathtt{n},\ \eta(q_i, i) = \mathtt{y},\ \eta(q_i, \bar{i}) = \mathtt{n};$$

each for $i \in \{0,1\}$, with $\bar{0} = 1$ and $\bar{1} = 0$. If this machine starts in q_ε, it outputs `y` or `n` depending on whether the last two symbols seen are the same or not, respectively. Thus, on input `011`, it has run $q_\varepsilon, q_0, q_1, q_1$ and output `nny`. ◇

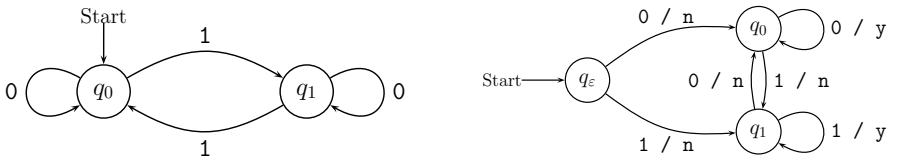


Fig. 8. (left) Finite automaton for parity, (right) Example Mealy machine

3.2 Adding continuous phenomena

In moving toward hybrid systems, we may add a variety of more complex continuous phenomena to the finite and discrete automata above. The continuous phenomena we will add are as follows:

- *Global time*, where one adds a single global clock with unity rate.
- *Timed automata*, where one adds a set of such clocks, and the ability to reset them,
- *Skewed-clock automata*, where each clock has a different, uniform (in each location), rational rate.
- *Multi-rate automata*, where each clock variable can take on different, rational rates in each location.

Next, we examine each of these in turn. For ease of discussion, note that discrete states are also referred to as *modes*, *phases*, or *locations*.

Global time

Usually, digital automata are thought of as evolving in "abstract time," where only the ordering of symbols or "events" matters. See (8). We may add the notion of time by associating with the kth transition the time, t_k, at which it occurs [15]:

$$\begin{aligned} q(t_{k+1}) &= \nu(q(t_k), i(t_k)), \\ o(t_k) &= \eta(q(t_k), i(t_k)). \end{aligned}$$

Finally, this automaton may be thought of as operating in "continuous time" by the convention that the state, input, and output symbols are piecewise continuous functions. We may again use Sontag's transition notation [31] to denote this. The result is the following system of equations, where q, o are piecewise continuous in time:

$$\begin{aligned} q^+(t) &= \nu(q(t), i(t)), \\ o(t) &= \eta(q(t), i(t)). \end{aligned} \tag{9}$$

Here, the state $q(t)$ changes only when the input symbol $i(t)$ changes.

Timed automata

Timed automata more fully incorporate the notion of real time with automata [1]. Whereas finite automata process words, timed automata process *timed* words. A *timed word* (over input alphabet I) is a finite sequence of symbols and their respective times of occurrence: $(i_1, t_1), (i_2, t_2), \ldots, (i_N, t_N)$, where each $i_k \in I$, each $t_k \in \mathbf{R}_+$, and times *strictly increase*: $t_{k+1} > t_k$. A *timed ω-word* is an infinite sequence $(i_1, t_1), (i_2, t_2), \ldots$, with all as above plus the

constraint that time *progresses* without bound: for all $T \in \mathbf{R}_+$, there exists a k such that $t_k > T$. The latter condition is necessary to avoid *Zeno behavior*. For example, the sequence of times $1/2, 3/4, 7/8, 15/16, 31/32, \ldots$ is not a valid time sequence. A *timed language* is merely a set of timed words.

A *timed automaton* is the same as an automaton except that one adds a finite number of real-valued clocks, plus the ability to reset clocks and test constraints on clocks when traversing edges. A *clock constraint*, χ, may take one of the forms

$$(x \leq c), \qquad (c \leq x), \qquad \neg\chi_0, \qquad \chi_1 \wedge \chi_2,$$

where x is a clock variable, c is a rational constant, $\neg$ denotes logical negation, $\wedge$ denotes logical "and", and χ_i are themselves valid clock constraints. Note that these forms plus the rules of logic allow one to build up more complicated tests, including the following:

$$\begin{aligned}
(x = c) &\Longleftrightarrow (x \leq c) \wedge (c \leq x),\\
(x < c) &\Longleftrightarrow (x \leq c) \wedge \neg(x = c),\\
\chi_1 \vee \chi_2 &\Longleftrightarrow \neg(\neg\chi_1 \wedge \neg\chi_2),\\
\mathsf{True} &\Longleftrightarrow (x \leq c) \vee (c \leq x).
\end{aligned}$$

In drawings, the notation $?\chi$ is used to denote the fact that the edge *may* be taken only if the clock constraint χ is true; we have found it useful to let $!\chi$ mean that the edge *must* be taken when χ is true.

Example 12 (Bounded Response Time). Consider the timed automaton in Fig. 9(left), which is taken from [1]. If it starts in q_0, it will process strings of the form $(\mathtt{ad})^\omega$ such that the time between every `a` and its corresponding `d` is less than 2. It can be used to model the fact that every "arrival" needs to be serviced (needs to "depart") within two seconds. ◇

Example 13 (Switch with Delay). This example is modified from one describing a metal oxide semiconductor (MOS) transistor [26]. It can also be used to model relays, pneumatic valves, and other switches with delay. Specifically, consider a switch that can be On or Off, but that takes one second to activate. Let `U` and `D` denote the symbolic inputs corresponding to commanding the switch on or off, respectively. Then the switch can be modeled as a timed automaton as in Fig. 9(right). Note that issuing command `U` resets the clock to zero, and if a `D` command arrives within one time unit later, the switch goes back to off. ◇

The theory of timed automata is almost as rich and beautiful as the theory of finite automata. In fact, certain verification problems for timed automata can be translated directly into questions about (generally much larger) finite-state machines. See [1] for details.

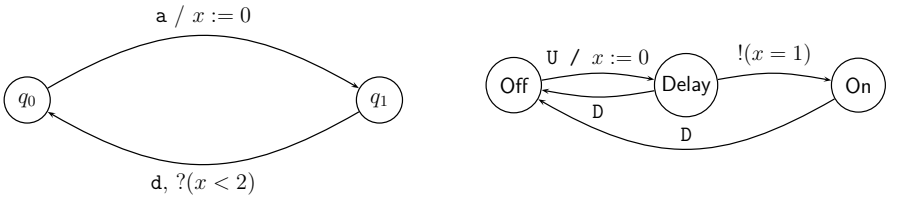


Fig. 9. Timed automata modeling (left) bounded response time, (right) a switch with delay

Skewed-clock automata

To summarize, a timed automaton has a finite set, C, of clocks, $x_i \in C$ such that $\dot{x_i} = 1$ for all clocks and all locations. In a *skewed-clock automaton*, $\dot{x}_i = k_i$ where each k_i is a rational number.

Remark 1. Skewed-clock automata are equivalent to timed automata. That is, any skewed-clock automaton can be converted into an equivalent timed automaton, and vice versa.

Proof. It is obvious that every timed automaton is a special case of a skewed-clock automaton, wherein each $k_i = 1$. For the converse, we have two cases to consider:

1. $k_i = 0$: In this case, $x_i(t)$ remains constant and any conditions involving it are uniformly true or false (and thus may be reduced or removed using the rules of logic).
2. $k_i \neq 0$: In this case, we note that $x_i(t) = x_i(0) + k_i t$, so that $x_i(t)/k_i = x_i(0)/k_i + t$. This means we can divide every constant to which x_i is compared by k_i, and then use the associated clock $\tilde{x}_i = x_i/k_i$, with $\dot{\tilde{x}}_i = 1$.

□

Multi-rate automata

A *multi-rate automaton* has $\dot{x}_i = k_{i,q}$ at location $q \in Q$, where each $k_{i,q}$ is a rational number; see Fig. 10.

Some notes are in order, both with respect to Fig. 10 and in general:

- Some variables might have the same rates in all states, e.g., w.
- Some variables might be *stopwatches* (derivative either 0 or 1), measuring a particular time. For example, x is a stopwatch measuring the elapsed time spent in the upper left state.
- Not all dynamics change at every transition. For example, y changes dynamics in transitioning along Edges 2 and 3, but not along Edge 1.
- Every skewed-clock automaton is a multi-rate automaton that simply has $k_{i,q} = k_i$ for all $q \in Q$.

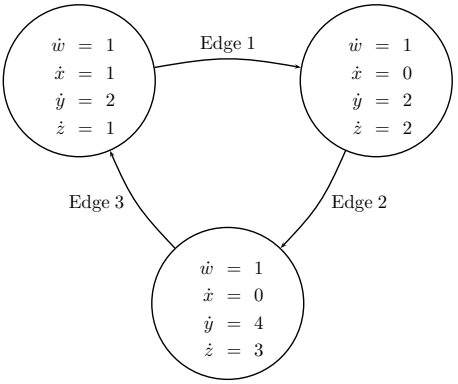


Fig. 10. Example multi-rate automaton (edge data has been suppressed)

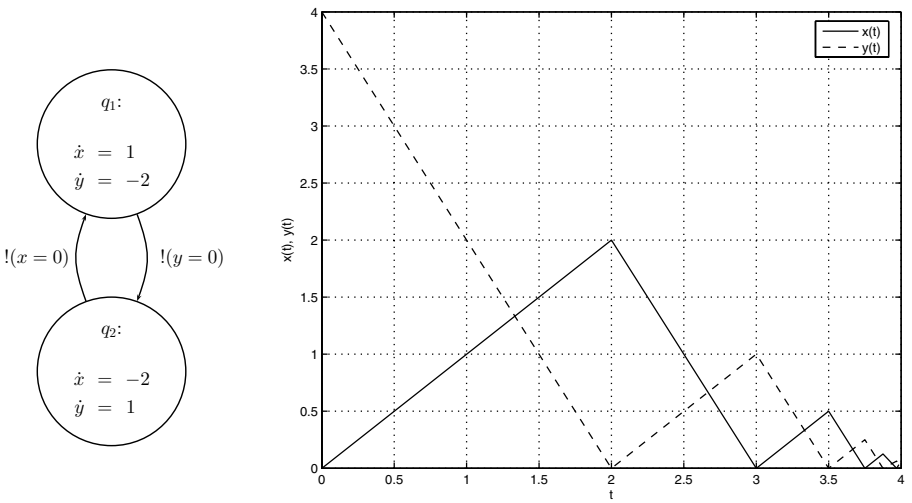


Fig. 11. (left) A multi-rate automaton with Zeno behavior. (right) Continuous-state dynamics over time, starting in q_1 at $(x, y) = (0, 4)$; events pile up at $t = 4$

Multi-rate automata can exhibit what is called *Zeno behavior* [39], where event times "pile up," not allowing time to progress; see Fig. 11. In general, the behavior of multi-rate automata is even more complicated, being computationally undecidable [21].

There is a simple constraint on multi-rate automata that yields a class that permits analysis. An *initialized* multi-rate automaton adds the constraint that a variable must be reset when traversing an edge if its dynamics change while crossing that edge. This was not the case for the Zeno system of Fig. 11. This would be the case for the automaton in Fig. 10 if x and z were initialized on Edge 1, y and z were initialized on Edge 2, and x, y, and z were initialized on Edge 3.

Remark 2. An initialized multi-rate automaton can be converted into a timed automaton.

Proof. The idea is to use the same trick as in Remark 1, as many times for each variable as it has different rates. Note that the fact that the automaton is "initialized" is crucial. See [21] for details. □

3.3 Other refinements

Rectangular automata: Going from rates to rectangular inclusions, one can define rectangular and multi-rectangular automata. Specifically, a *rectangular automaton* has each $\dot{x}_i \in [l_i, u_i]$, where l_i and u_i are rational constants, uniformly over all locations. Thus, they generalize skewed-clock automata. Indeed, letting $l_i = u_i = k_i$ recovers them. Analogously, multi-rate automata can be generalized to *multi-rectangular automata*. That is, each variable satisfies a (generally different) rectangular inclusion in each location. Usually, rectangular automata allow the setting of initial values and the resetting of variables to be performed within rectangles as well. *Initialized* multi-rectangular automata are constrained to perform a reset on a variable along edges that change the rectangular inclusions governing its dynamics.

Remark 3. An initialized multi-rectangular automaton can be converted to an initialized multi-rate automaton (and hence a timed automaton).

Proof. The idea is to replace each continuous variable, say x, with two variables, say x_l and x_u, that track lower and upper bounds on its value, respectively. See [21] for details. Then, invoke Remark 2. □

Adding control: A rich control theory for automata models can be built by allowing the disabling of certain *controllable* input symbols. See [15, 30].

4 Hybrid Dynamical Systems and Hybrid Automata

Briefly, a hybrid dynamical system is an indexed collection of ODEs along with a map for "jumping" among them (switching the dynamical system and/or resetting the state). This jumping occurs whenever the state satisfies certain conditions, given by its membership in a specified subset of the state space. Hence, the entire system can be thought of as a sequential patching together of ODEs with initial and final states, the jumps performing a reset to a (generally different) initial state of a (generally different) ODE whenever a final state is reached. See Fig. 12.

Formally, a *hybrid dynamical system* (*HDS*)[3] is a system $H = (Q, \mathbf{\Sigma}, \mathbf{A}, \mathbf{G})$, with constituent parts as follows:

[3] A more general notion, *GHDS* (*general HDS*), appears in [7]. It allows each ODE to be replaced by a general dynamical system; most notably, one could replace the ODEs with difference equations.

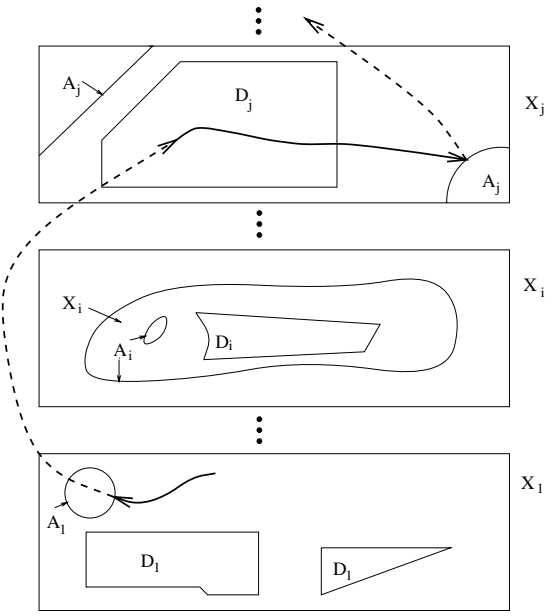


Fig. 12. Example dynamics of an HDS

- Q is the countable set of *index* or *discrete states*.
- $\mathbf{\Sigma} = \{\Sigma_q\}_{q\in Q}$ is the collection of *systems* of ODEs. Thus, each system Σ_q has vector fields $f_q : X_q \to \mathbf{R}^{d_q}$, representing the *continuous dynamics*, evolving on $X_q \subset \mathbf{R}^{d_q}$, $d_q \in \mathbf{Z}_+$, which are the *continuous state spaces*.
- $\mathbf{A} = \{A_q\}_{q\in Q}$, $A_q \subset X_q$ for each $q \in Q$, is the collection of *autonomous jump sets*.
- $\mathbf{G} = \{G_q\}_{q\in Q}$, where $G_q : A_q \to S$ are the *autonomous jump transition maps*, said to represent the *discrete dynamics*.

Thus, $S = \bigcup_{q\in Q} X_q \times \{q\}$ is the *hybrid state space* of H. For convenience, we use the following shorthand. $S_q = X_q \times \{q\}$, and $A = \bigcup_{q\in Q} A_q \times \{q\}$ is *the* autonomous jump set. $G : A \longrightarrow S$ is *the* autonomous jump transition map, constructed componentwise in the obvious way. The *jump destination sets* $\mathbf{D} = \{D_q\}_{q\in Q}$ may be defined by $D_q = \pi_1[G(A) \cap S_q]$, where π_i is projection onto the ith coordinate. The *switching manifolds* or *transition manifolds*, $M_{q,p} \subset A_q$, are given by $M_{q,p} = G_q^{-1}(D_p, p)$, i.e., the set of states for which transitions from index q to index p can occur.

The dynamics of the HDS H are as follows.[4] The system is assumed to start in some hybrid state in $S\backslash A$, say $s_0 = (x_0, q_0)$. It evolves according to $\dot{x} = f_{q_0}(x)$ until the state enters—if ever—A_{q_0} at the point $s_1^- = (x_1^-, q_0)$. At

[4]For the solutions described to be well defined, it is sufficient that for all q, A_q is closed, $D_q \cap A_q = \emptyset$, and f_q is Lipschitz continuous. Note, however, that in general these solutions are not continuous in the initial conditions. See [7] for details.

this time it is instantly transferred according to transition map to $G_{q_0}(x_1^-) = (x_1, q_1) \equiv s_1$, from which the process continues.

We now collect some notes about HDS:

- *ODEs and automata.* First, note that in the case $|Q| = 1$ and $A = \emptyset$ we recover all differential equations. With $|Q| = N$ and each $f_q \equiv 0$, we recover finite automata.
- *Outputs.* It is straightforward to add continuous and discrete output maps for each constituent system to the above model.
- *Changing state space and overlaps.* The state space may change. This is useful in modeling component failures or changes in dynamical description based on autonomous—and later, controlled—events which change it. Examples include the collision of two inelastic particles or an aircraft mode transition that changes variables to be controlled [27]. We allow the X_q to overlap and the inclusion of multiple copies of the same space. This may be used, for example, to take into account overlapping local coordinate systems on a manifold [4]. It was also used in the hysteresis example.
- *Transition delays.* It is possible to model the fact that autonomous jumps may not be instantaneous by simply adding an *autonomous jump delay map*, $\Delta_a : A \times V \longrightarrow \mathbf{R}_+$. This map associates a (possibly zero) delay to each autonomous jump. Thus, a jump begins at some time, τ, and is completed at some later time $\Gamma \geq \tau$. This concept is useful as jumps may be introduced to represent an aggregate set of relatively fast, transitory dynamics. Also, some commands include a finite delay from issuance to completion. An example is closing a hydraulic valve.

It is easy to see that the case of HDS with $|Q|$ finite is a coupling of finite automata and differential equations. Indeed, an HDS can be pictured as a *hybrid automaton* as in Fig. 13. There, each node is a constituent ODE, with the index the name of the node. Each edge represents a possible transition between constituent systems, labeled by the appropriate condition for the transition's being "enabled" and the update of the continuous state (cf. [20]). The notation ![condition] denotes that the transition *must* be taken when enabled.

Example 14 (Bouncing Ball Revisited). We may draw a hybrid automaton associated with the bouncing ball of Example 4. The velocity resets are autonomous and must occur when the ball hits the ground. See Fig. 14. ◇

Example 15 (Furnace Revisited). Consider the furnace controller of Example 2, and a goal to keep the temperature around 23 degrees Celsius. To avoid inordinate switching around the setpoint (known as *chattering*), we implement the control with hysteresis as in Fig. 15(left). This controller will

1. Turn the furnace On when it is Off and the temperature falls below 21.
2. Turn the furnace Off when it is On and the temperature rises above 25.

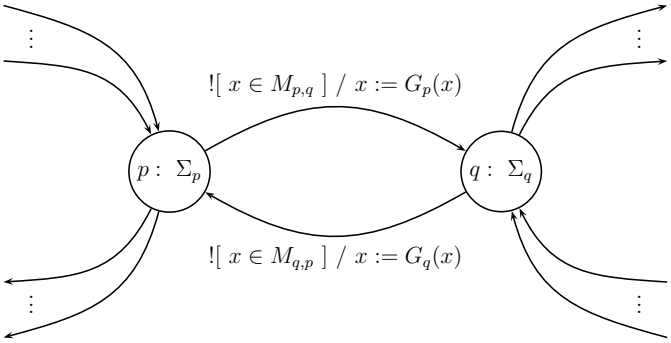


Fig. 13. Hybrid automaton representation of an HDS

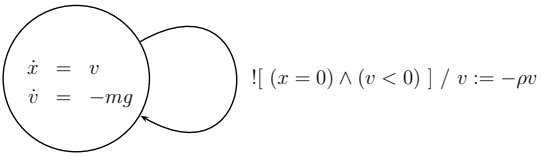


Fig. 14. Hybrid automaton for the bouncing ball of Example 4

Suppose, also, that the furnace should never be on more than 55 minutes straight, and that it must remain Off for at least 5 minutes. The latter can be accomplished by adding a timer variable that is reset on state transitions. The refined hybrid controller is pictured in Fig. 15(right). ◇

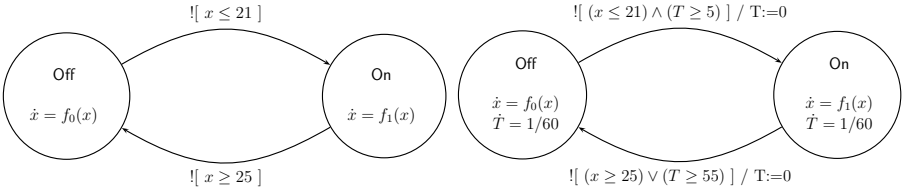


Fig. 15. Hybrid controllers for furnace: (left) simple hysteresis, (right) refinement

4.1 Adding control

We now add to the above the ability to make decisions, that is, to control an HDS by choosing among sets of possible actions at various times. Specifically, we allow (i) the possibility of continuous controls for each ODE, (ii) the ability to make decisions at the autonomous jump points, and (iii) to add controlled jumps, where one may decide to jump or not and have a choice of destination when the state satisfies certain constraints.

Formally, a *controlled hybrid dynamical system* (*CHDS*) is a system $H_c = (Q, \mathbf{\Sigma}, \mathbf{A}, \mathbf{G}, \mathbf{C}, \mathbf{F})$, with constituent parts as in an HDS, except as follows:

- $\mathbf{\Sigma} = \{\Sigma_q\}_{q\in Q}$ is now a collection of *controlled* ODEs, with controlled vector fields $f_q : X_q \times U_q \to \mathbf{R}^{d_q}$ representing the (controlled) *continuous dynamics*, where $U_q \subset \mathbf{R}^{m_q}$, $m_q \in \mathbf{Z}_+$, are the *continuous control spaces*.
- $\mathbf{G} = \{G_q\}_{q\in Q}$, where $G_q : A_q \times V_q \to S$ are the *autonomous jump transition maps*, now modulated by the *discrete decision sets* V_q.
- $\mathbf{C} = \{C_q\}_{q\in Q}$, $C_q \subset X_q$, is the collection of *controlled jump sets*.
- $\mathbf{F} = \{F_q\}_{q\in Q}$, where $F_q : C_q \to 2^S$, is the collection of set-valued *controlled jump destination maps*.

Again, $S = \bigcup_{q\in Q} X_q \times \{q\}$ is the hybrid state space of H_c. As before, we introduce some shorthand beyond that defined for the HDS above. We let $C = \bigcup_{q\in Q} C_q \times \{q\}$; we let $F : C \to 2^S$ denote *the* set-valued map composed in the obvious way from the set-valued maps F_q.

The dynamics of the CHDS H_c are as follows. The system is assumed to start in some hybrid state in $S\backslash A$, say $s_0 = (x_0, q_0)$. It evolves according to $\dot{x} = f_{q_0}(x, u)$ until the state enters—if ever—either A_{q_0} or C_{q_0} at the point $s_1^- = (x_1^-, q_0)$. If it enters A_{q_0}, then it *must* be transferred according to transition map $G_{q_0}(x_1^-, v)$ for some chosen $v \in V_{q_0}$. If it enters C_{q_0}, then we *may* choose to jump and, if so, we may choose the destination to be any point in $F_{q_0}(x_1^-)$. In either case, we arrive at a point $s_1 = (x_1, q_1)$ from which the process continues. See Fig. 16.

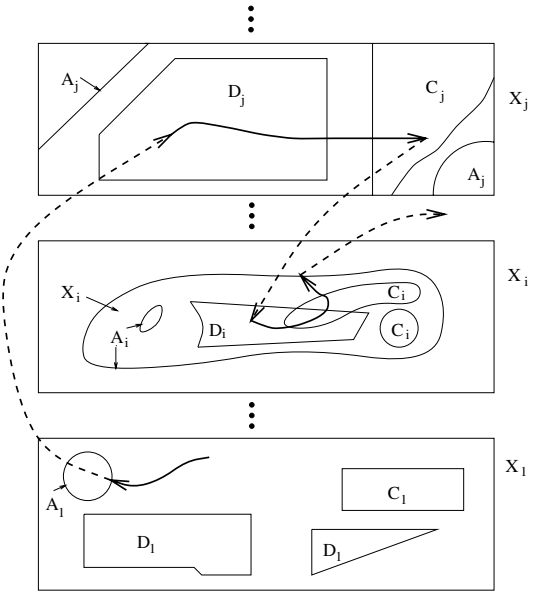


Fig. 16. Example dynamics of a CHDS

A CHDS can also be pictured as an automaton, as in Fig. 17. There, each node is a constituent controlled ODE, with the index the name of the node. The notation ?[condition] denotes an enabled transition that *may* be taken on command; ":∈" means reassignment to some value in the given set.

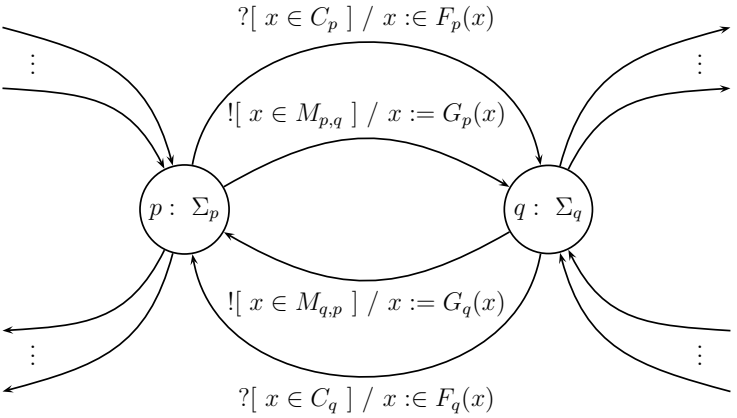


Fig. 17. Hybrid automaton representation of a CHDS

Example 16 (Transmission Revisited). Some modern performance sedans offer the driver the ability to shift gears electronically. However, there are often rules in place that prevent certain shifts, or automatically perform certain shifts, to maintain safe operation and reduce wear. These rules are often a function of the engine RPM (x_2 in Example 6). A portion of the hybrid controller incorporating such logic is shown in Fig. 18. The rules pictured only allow the driver to shift from Gear 1 to Gear 2 if the RPM exceeds 1800, but automatically makes this switch if the RPM exceeds 3500. Similar rules would exist for higher gears; more complicated rules might exist regarding Neutral and Reverse. ◇

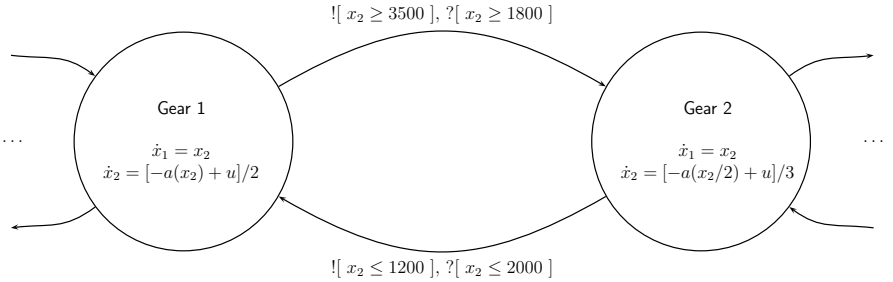


Fig. 18. Portion of hybrid transmission controller

5 Going Further

A survey of the hybrid systems literature is well beyond the scope of this chapter. For early surveys and more details on hybrid systems modeling, see [2,7,19]. For a recent monograph on switching systems, see [24]. Analysis and control techniques for hybrid systems have been developed. See [7] for details and [9] for a summary. For a more complete survey of stability tools, see [8,16]. The papers [10,11,13] developed an optimal control theory and algorithms for designing hybrid control systems. A game theoretic approach appears in [34]. For an introduction to hybrid systems simulation, see [12].

References

1. Alur, R., and Dill, D.L. (1994) A theory of timed automata. *Theoretical Computer Science*, 126:183–235.
2. Alur, R., Courcoubetis, C., Henzinger, T.A., and Ho, P.-H. (1993) Hybrid automata: An algorithmic approach to the specification and verification of hybrid systems. In: Grossman, R., Nerode, A., Ravn, A., and Rischel, H. (eds) (1993) *Hybrid Systems*, pp. 209–229. Springer, New York.
3. Antsaklis, P.J., Stiver, J.A., and Lemmon, M.D. (1993) Hybrid system modeling and autonomous control systems. In: Grossman, R., Nerode, A., Ravn, A., and Rischel, H. (eds) (1993) *Hybrid Systems*, pp. 366–392. Springer, New York.
4. Back, A., Guckenheimer, J., and Myers, M. (1993) A dynamical simulation facility for hybrid systems. In: Grossman, R., Nerode, A., Ravn, A., and Rischel, H. (eds) (1993) *Hybrid Systems*, pp. 255–267. Springer, New York.
5. Bainov, D.D., and Simeonov, P.S. (1989) *Systems with Impulse Effect.* Ellis Horwood, Chichester, England.
6. Bensoussan, A., and Lions, J.-L. (1984) *Impulse Control and Quasi-Variational Inequalities.* Gauthier-Villars, Paris.
7. Branicky, M.S. (1995) *Studies in Hybrid Systems: Modeling, Analysis, and Control.* ScD thesis, Massachusetts Institute of Technology, Cambridge, MA.
8. Branicky, M.S. (1997) Stability of hybrid systems: state of the art. In: *Proc. IEEE Conf. Decision and Control*, pp. 120–125, San Diego, CA.
9. Branicky, M.S. (1998) Analyzing and synthesizing hybrid control systems. In: Rozenberg, G., and Vaandrager, F. (eds) *Lectures on Embedded Systems*, pp. 74–113. Springer, Berlin.
10. Branicky, M.S., Borkar, V.S., and Mitter, S.K. (1998) A unified framework for hybrid control: Model and optimal control theory. *IEEE Trans. Automatic Control*, 43(1):31–45.
11. Branicky, M.S., Hebbar, R., and Zhang, G. (1999) A fast marching algorithm for hybrid systems. In: *Proc. IEEE Conf. Decision and Control*, pp. 4897–4902. Phoenix, AZ.
12. Branicky, M.S., and Mattsson, S.E. (1997) Simulation of hybrid systems. In: Antsaklis, P.J., Kohn, W., Nerode, A., and Sastry, S. (eds) *Hybrid Systems IV*, pp. 31–56. Springer, New York.
13. Branicky, M.S., and Mitter, S.K. (1995) Algorithms for optimal hybrid control. *Proc. IEEE Conf. Decision and Control*, pp. 2661–2666, New Orleans, LA.

14. Brockett, R.W. (1993) Hybrid models for motion control systems. In: Trentelman, H.L., and Willems, J.C. (eds) *Essays in Control*, pp. 29–53. Birkhäuser, Boston.
15. Cassandras, C.G., and Lafortune, S. (1999) *Introduction to Discrete Event Systems*. Kluwer Academic Publishers, Boston.
16. DeCarlo, R., Branicky, M.S., Pettersson, S., and Lennartson, B. (2000) Perspectives and results on the stability and stabilizability of hybrid systems. *Proc. IEEE*, 88(2):1069–1082.
17. Ghosh, R., and Tomlin, C. (2004) Symbolic reachable set computation of piecewise affine hybrid automata and its application to biological modelling: delta-notch protein signalling. *Systems Biology*, 1(1):170–183.
18. Gollu, A., and Varaiya, P.P. (1989) Hybrid dynamical systems. In: *Proc. IEEE Conf. Decision and Control*, pp. 2708–2712. Tampa, FL.
19. Grossman, R., Nerode, A., Ravn, A., and Rischel, H. (eds) (1993) *Hybrid Systems*. Springer, New York.
20. Harel, D. (1987) Statecharts: A visual formalism for complex systems. *Science Computer Programming*, 8:231–274.
21. Henzinger, T.A., Kopke, P.W., Puri, A., and Varaiya, P. (1998) What's decidable about hybrid automata? *J. Computer and System Sciences*, 57:94–124.
22. Hirsch, M.W., and Smale, S. (1974) *Differential Equations, Dynamical Systems, and Linear Algebra*. Academic Press, San Diego, CA.
23. Hopcroft, J.E., and Ullman, J.D. (1979) *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, Reading, MA.
24. Liberzon, D. (2003) *Switching in Systems and Control*. Birkhauser, Boston.
25. Luenberger, D.G. (1979) *Introduction to Dynamic Systems*. Wiley, New York.
26. Maler, O., and Yovine, S. (1996) Hardware timing verification using KRONOS. In: *Proc. 7th Israeli Conf. Computer Systems and Software Eng.* Herzliya, Israel.
27. Meyer, G. (1994) Design of flight vehicle management systems. In: *IEEE Conf. Decision and Control*, Plenary Lecture. Lake Buena Vista, FL.
28. Nerode, A., and Kohn, W. (1993) Models for hybrid systems: Automata, topologies, controllability, observability. In: Grossman, R., Nerode, A., Ravn, A., and Rischel, H. (eds) (1993) *Hybrid Systems*, pp. 317–356. Springer, New York.
29. Raibert, M.H. (1986) *Legged Robots that Balance*. MIT Press, Cambridge, MA.
30. Ramadge, P.J.G., and Wonham, M.W. (1989) The control of discrete event systems. *Proc. IEEE*, 77(1):81–98.
31. Sontag, E.D. (1990) *Mathematical Control Theory*. Springer, New York.
32. Tabuada, P., Pappas, G.J., and Lima, P. (2001) Feasible formations of multi-agent systems. In: *Proc. Amer. Control Conf.*, Arlington, VA.
33. Tavernini, L. (1987) Differential automata and their discrete simulators. *Nonlinear Analysis, Theory, Methods, and Applications*, 11(6):665–683.
34. Tomlin, C., Lygeros, J., and Sastry, S. (2000) A game theoretic approach to controller design for hybrid systems. *Proc. IEEE*, 88(7):949–970.
35. Tomlin, C., Pappas, G.J., and Sastry, S. (1998) Conflict resolution for air traffic management: A study in multi-agent hybrid systems. *IEEE Trans. Automatic Control*, 43(4):509–521.
36. Varaiya, P.P. (1993) Smart cars on smart roads: Problems of control. *IEEE Trans. Automatic Control*, 38(2):195–207.
37. Witsenhausen, H.S. (1966) A class of hybrid-state continuous-time dynamic systems. *IEEE Trans. Automatic Control*, AC-11(2):161–167.

38. Zabczyk, J. (1973) Optimal control by means of switching. *Studia Mathematica*, 65:161–171.
39. Zhang, J., Johansson, K.H., Lygeros, J., and Sastry, S. (2000) Dynamical systems revisited: Hybrid systems with Zeno executions. In: Lynch, N., and Krogh, B. (eds) *Hybrid Systems: Computation and Control*, pp. 451–464. Springer, Berlin.
40. Zhang, W., Branicky, M.S., and Phillips, S.M. (2001) Stability of networked control systems. *IEEE Control Systems Magazine*, 21(1):84–99.

Finite Automata

M. V. Lawson

Department of Mathematics
School of Mathematical and Computer Sciences
Heriot-Watt University
Riccarton, Edinburgh EH14 4AS
Scotland
M.V.Lawson@ma.hw.ac.uk

1 Introduction

The term 'finite automata' describes a class of models of computation that are characterised by having a finite number of states. The use of the word 'automata' harks back to the early days of the subject in the 1950's when they were viewed as abstract models of real circuits. However, the readers of this chapter can also view them as being implemented by programming languages, or as themselves constituting a special class of programming languages. Taking this latter viewpoint, finite automata form a class of programming languages with very restricted operation sets: roughly speaking, programs where there is no recursion or looping. In particular, none of them is a universal programming language; indeed, it is possible to prove that some quite straightforward operations cannot be programmed even with the most general automata I discuss. Why, then, should anyone be interested in programming with, as it were, one hand tied behind his back? The answer is that automata turn out to be useful — so they are not merely mathematical curiosities. In addition, because they are restricted in what they can do, we can actually say more about them, which in turn helps us manipulate them.

The most general model I discuss in this chapter is that of a finite transducer, in Section 3.3, but I build up to this model in Sections 2, 3.1, and 3.2 by discussing, in increasing generality: finite acceptors, finite purely sequential transducers, and finite sequential transducers. The finite acceptors form the foundation of the whole enterprise through their intimate link with regular expressions and, in addition, they form the pattern after which the more general theories are modeled.

What then are the advantages of the various kinds of finite transducers considered in this chapter? There are two main ones: the speed with which data can be processed by such a device, and the algorithms that enable one to make the devices more efficient. The fact that finite transducers of vari-

ous kinds have turned out to be useful in natural language processing is a testament to both of these advantages [23]. I discuss the advantages of finite transducers in a little more detail in Section 4.

To read this chapter, I have assumed that you have been exposed to a first course in discrete math(s); you need to know a little about sets, functions, and relations, but not much more. My goal has been to describe the core of the theory in the belief that once the basic ideas have been grasped, the business of adding various bells and whistles can easily be carried out according to taste.

Other reading There are two classic books outlining the theory of finite transducers: Berstel [4] and Eilenberg [12]. Of these, I find Berstel's 1979 book the more accessible. However, the theory has moved on since 1979, and in the course of this chapter I refer to recent papers that take the subject up to the present day. In particular, the paper [24] contains a modern, mathematical approach to the basic theory of finite transducers. The book by Jacques Sakarovitch [30] is a recent account of automata theory that is likely to become a standard reference. The chapters by Berstel and Perrin [6], on algorithms on words, and by Laporte [21], on symbolic natural language processing, both to be found in the forthcoming book by M. Lothaire, are excellent introductions to finite automata and their applications. The articles [25] and [35] are interesting in themselves and useful for their lengthy bibliographies. My chapter deals entirely with finite strings — for the theory of infinite strings see [26]. Finally, finite transducers are merely a part of theoretical computer science; for the big picture, see [17].

Terminology This has not been entirely standardised so readers should be on their guard when reading papers and books on finite automata. Throughout this chapter I have adopted the following terminology introduced by Jacques Sakarovitch and suggested to me by Jean-Eric Pin: a 'purely sequential function' is what is frequently referred to in the literature as a 'sequential function'; whereas a 'sequential function' is what is frequently referred to as a 'subsequential function'. The new terminology is more logical than the old, and signals more clearly the role of sequential functions (in the new sense).

2 Finite Acceptors

The automata in this section might initially not seem very useful: their response to an input is to output either a 'yes' or a 'no'. However, the concepts and ideas introduced here provide the foundation for the rest of the chapter, and a model for the sorts of things that finite automata can do.

2.1 Alphabets, strings, and languages

Information is encoded by means of sequences of symbols. Any finite set A used to encode information will be called an *alphabet*, and any finite sequence whose components are drawn from A is called a *string over* A or simply a *string*, and sometimes a *word*. We call the elements of an alphabet *symbols*, *letters*, or *tokens*. The symbols in an alphabet do not have to be especially simple; an alphabet could consist of pictures, or each element of an alphabet could itself be a sequence of symbols. A string is formally written using brackets and commas to separate components. Thus (now, is, the, winter, of, our, discontent) is a string over the alphabet whose symbols are the words in an English dictionary. The string () is the empty string. However, we shall write strings without brackets and commas and so, for instance, we write 01110 rather than $(0, 1, 1, 1, 0)$. The empty string needs to be recorded in some way and we denote it by ε. The set of all strings over the alphabet A is denoted by A^*, read 'A star'. If w is a string then $|\, w \,|$ denotes the total number of symbols appearing in w and is called the *length of* w. Observe that $|\, \varepsilon \,| = 0$. It is worth noting that two strings u and v over an alphabet A are *equal* if they contain the same symbols in the same order.

Given two strings $x, y \in A^*$, we can form a new string $x \cdot y$, called the *concatenation of* x *and* y, by simply adjoining the symbols in y to those in x. We shall usually denote the concatenation of x and y by xy rather than $x \cdot y$. The string ε has a special property with respect to concatenation: for each string $x \in A^*$ we clearly have that $\varepsilon x = x = x\varepsilon$. It is important to emphasise that the order in which strings are concatenated is important: thus xy is generally different from yx.

There are many definitions concerned with strings, but for this chapter I just need two. Let $x, y \in A^*$. If $u = xy$ then x is called a *prefix* of u, and y is called a *suffix* of u.

Alphabets and strings are needed to define the key concept of this section: that of a language. Before formally defining this term, here is a motivating example.

Example 1. Let A be the alphabet that consists of all words in an English dictionary; so we regard each English word as being a single symbol. The set A^* consists of all possible finite sequences of words. Define the subset L of A^* to consist of all sequences of words that form grammatically correct English sentences. Thus

$$\text{to be or not to be} \in L$$

whereas

$$\text{be be to to or not} \notin L.$$

Someone who wants to understand English has to learn the rules for deciding when a string of words belongs to the set L. We can therefore think of L as being the English language.

For *any* alphabet A, *any* subset of A^* is called an *A-language* or a *language over A* or simply a *language*. Languages are usually infinite; the question we shall address in Section 2.2 is to find a finite way of describing (some) infinite languages.

There are a number of ways of combining languages to make new ones. If L and M are languages over A so are $L \cap M$, $L \cup M$, and L': respectively, the intersection of L and M, the union of L and M, and the complement of L. These are called the *Boolean operations* and come from set theory. Recall that '$x \in L \cup M$' means '$x \in L$ or $x \in M$ or both.' In automata theory, we usually write $L + M$ rather than $L \cup M$ when dealing with languages. There are two further operations on languages that are peculiar to automata theory and extremely important: the product and the Kleene star. Let L and M be languages. Then

$$L \cdot M = \{xy\colon\ x \in L \text{ and } y \in M\}$$

is called the *product of L and M*. We usually write LM rather than $L \cdot M$. A string belongs to LM if it can be written as a string in L *followed by* a string in M. For a language L, we define $L^0 = \{\varepsilon\}$, and $L^{n+1} = L^n \cdot L$. For $n > 0$, the language L^n consists of all strings u of the form $u = x_1 \dots x_n$ where $x_i \in L$. The *Kleene star* of a language L, denoted L^*, is defined to be

$$L^* = L^0 + L^1 + L^2 + \dots .$$

2.2 Finite acceptors

An information-processing device transforms inputs into outputs. In general, there are two alphabets associated with such a device: an *input alphabet* A for communicating with it, and an *output alphabet* B for receiving answers. For example, consider a device that takes as input sentences in English and outputs the corresponding sentence in Russian. In later sections, I shall describe mathematical models of such devices of increasing generality. In this section, I shall look at a special case: there is an input alphabet A, but each input string causes the device to output either 'yes' or 'no' once the whole input has been processed. Those input strings from A^* that cause the machine to output 'yes' are said to be *accepted* by the machine, and those that cause it to output 'no' are said to be *rejected*. In this way, A^* is partitioned into two subsets: the 'yes' subset we call the *language accepted by the machine*, and the 'no' subset we call the *language rejected by the machine*. A device that operates in this way is called an *acceptor*. We shall describe a mathematical model of a special class of acceptors. Our goal is *to describe potentially infinite languages by finite means*.

A *finite (deterministic) acceptor* **A** is specified by five pieces of information:

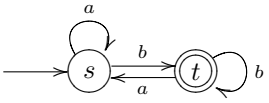
$$\mathbf{A} = (S, A, i, \delta, T)\,,$$

where S is a finite set called the *set of states*, A is the finite *input alphabet*, i is a fixed element of S called the *initial state*, δ is a function $\delta\colon\ S \times A \to S$,

called the *transition function*, and T is a subset of S (possibly empty!) called the set of *terminal* or *final states*.

There are two ways of providing the five pieces of information needed to specify an acceptor: transition diagrams and transition tables. A *transition diagram* is a special kind of directed labeled graph: the vertices are labeled by the states S of $\mathbf{A}$; there is an arrow labeled a from the vertex labeled s to the vertex labeled t precisely when $\delta(s, a) = t$ in $\mathbf{A}$. That is to say, the input a causes the acceptor $\mathbf{A}$ to change from state s to state t. Finally, the initial state and terminal states are distinguished in some way: we mark the initial state by an inward-pointing arrow, $\longrightarrow(i)$, and the terminal states by double circles $\textcircled{\textcircled{t}}$. A *transition table* is just a way of describing the transition function δ in tabular form and making clear in some way the initial and terminal states. The table has rows labeled by the states and columns labeled by the input letters. At the intersection of row s and column a we put the element $\delta(s, a)$. The states labeling the rows are marked to indicate the initial state and the terminal states.

Example 2. Here is a simple example of a transition diagram of a finite acceptor.



We can easily read off the five ingredients that specify an acceptor from this diagram:

- The set of states is $S = \{s, t\}$.
- The input alphabet is $A = \{a, b\}$.
- The initial state is s.
- The set of terminal states is $\{t\}$.

The transition function $\delta\colon\ S \times A \to S$ is given by

$$\delta(s, a) = s, \quad \delta(s, b) = t, \quad \delta(t, a) = s, \text{ and } \delta(t, b) = t.$$

Here is the transition table of our acceptor.

	a	b
$\to s$	s	t
$\leftarrow t$	s	t

We designate the initial state by an inward-pointing arrow $\to$ and the terminal states by outward-pointing arrows $\leftarrow$. If a state is both initial and terminal, then the inward- and outward-pointing arrows will be written as a single double-headed arrow $\leftrightarrow$.

To avoid too many arrows cluttering up a transition diagram, the following convention is used: if the letters $a_1, \ldots, a_m$ label m transitions from the state s to the state t, then we simply draw *one* arrow from s to t labeled $a_1, \ldots, a_m$ rather than m arrows labeled a_1 to a_m, respectively.

Let $\mathbf{A}$ be a finite acceptor with input alphabet A and initial state i. For each state q of $\mathbf{A}$ and for each input string x, there is a unique path in $\mathbf{A}$ that begins at q and is labeled by the symbols in x in turn. This path ends at a state we denote by $q \cdot x$. We say that x is *accepted* by $\mathbf{A}$ if $i \cdot x$ is a terminal state. That is, x labels a path in $\mathbf{A}$ that begins at the initial state and ends at a terminal state. Define the *language accepted* or *recognised* by $\mathbf{A}$, denoted $L(\mathbf{A})$, to be the set of all strings in the input alphabet that are accepted by $\mathbf{A}$. A language is said to be *recognisable* if it is accepted by some finite automaton. Observe that the empty string is accepted by an automaton if and only if the initial state is also terminal.

Example 3. We describe the language recognised by our acceptor in Example 2. We have to find all those strings in $(a+b)^*$ that label paths starting at s and finishing at t. First, any string x ending in a 'b' will be accepted. To see why, let $x = x'b$ where $x' \in A^*$. If x' leads the acceptor to state s, then the b will lead the acceptor to state t; and if x' leads the acceptor to state t, then the b will keep it there. Second, a string x ending in 'a' will not be accepted. To see why, let $x = x'a$ where $x' \in A^*$. If x' leads the acceptor to state s, then the a will keep it there; and if x' leads the acceptor to state t, then the a will send it to state s. We conclude that $L(\mathbf{A}) = A^*\{b\}$.

Here are some further examples of recognisable languages. I leave it as an exercise to the reader to construct suitable finite acceptors.

Example 4. Let $A = \{a, b\}$.

(i) The empty set $\emptyset$ is recognisable.
(ii) The language $\{\varepsilon\}$ is recognisable.
(iii) The languages $\{a\}$ and $\{b\}$ are recognisable.

It is worth pointing out that not all languages are recognisable. For example, the language consisting of those strings of a's and b's having an equal number of a's and b's is not recognisable.

One very important feature of finite (deterministic) acceptors needs to be highlighted, since it has great practical importance. The time taken for a finite acceptor to determine whether a string is accepted or rejected is a linear function of the length of the string; once a string has been completely read, we will have our answer.

The classic account of the theory of finite acceptors and their languages is contained in the first three chapters of [16]. The first two chapters of my book [22] describe the basics of finite acceptors at a more elementary level.

2.3 Non-deterministic ε-acceptors

The task of constructing a finite acceptor to recognise a given language can be a frustrating one. The chief reason for the difficulty is that finite acceptors are quite rigid: they have *one* initial state, and for each input letter and each state exactly *one* transition. Our first step, then, will be to relax these two conditions.

A *finite non-deterministic acceptor* $\mathbf{A}$ is determined by five pieces of information:

$$\mathbf{A} = (S, A, I, \delta, T),$$

where S is a finite set of states, A is the input alphabet, I is a set of initial states, $\delta\colon\ S \times A \to \mathsf{P}(S)$ is the transition function, where $\mathsf{P}(S)$ is the set of all subsets of S, and T is a set of terminal states. In addition to allowing any number of initial states, the key feature of this definition is that $\delta(s, a)$ is now a subset of S, possibly empty. The transition diagrams and transition tables we defined for deterministic acceptors can easily be adapted to describe non-deterministic ones. If q is a state and x a string, then the set of all states q' for which there is a path in $\mathbf{A}$ beginning at q, ending at q', and labeled by x is denoted by $q \cdot x$. The language $L(\mathbf{A})$ recognised by a non-deterministic acceptor consists of all those strings in A^* that label a path in $\mathbf{A}$ from at least one of the initial states to at least one of the terminal states.

It might be thought that, because there is a degree of choice available, non-deterministic acceptors might be able to recognise languages that deterministic ones could not. In fact, this is not so.

Theorem 1. *Let* $\mathbf{A}$ *be a finite non-deterministic acceptor. Then there is an algorithm for constructing a deterministic acceptor,* $\mathbf{A}^d$*, such that* $L(\mathbf{A}^d) = L(\mathbf{A})$.

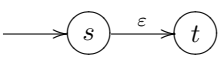
We now introduce a further measure of flexibility in constructing acceptors. In both deterministic and non-deterministic acceptors, transitions may only be labeled with elements of the input alphabet; no edge may be labeled with the empty string ε. We shall now waive this restriction. A *finite non-deterministic acceptor with ε-transitions* or, more simply, a *finite ε-acceptor*, is a 5-tuple,

$$\mathbf{A} = (S, A, I, \delta, T),$$

where all the symbols have the same meanings as in the non-deterministic case except that now

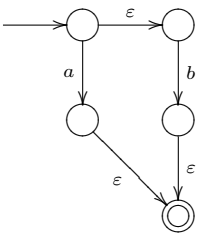
$$\delta\colon\ S \times (A \cup \{\varepsilon\}) \to \mathsf{P}(S).$$

The only difference between such acceptors and non-deterministic ones is that we allow transitions, called *ε-transitions*, of the form



A path in an ε-acceptor is a sequence of states each labeled by an element of the set $A \cup \{\varepsilon\}$. The string corresponding to this path is the *concatenation* of these labels in order; it is important to remember at this point that for every string x we have that $\varepsilon x = x = x\varepsilon$. We say that a string x is accepted by an ε-automaton if there is a path from an initial state to a terminal state the *concatenation of whose labels is* x.

Example 5. Consider the following finite ε-acceptor:

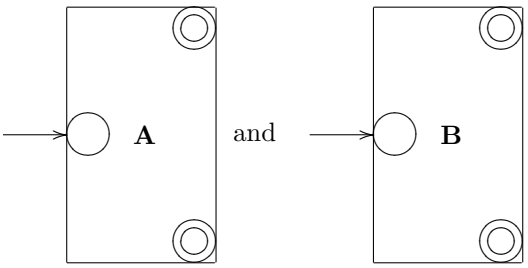


The language it recognises is $\{a, b\}$. The letter a is recognised because $a\varepsilon$ labels a path from the initial to the terminal state, and the letter b is recognised because $\varepsilon b\varepsilon$ labels a path from the initial to the terminal state.

The existence of ε-transitions introduces a further measure of flexibility in building acceptors but, as the following theorem shows, we can convert such acceptors to non-deterministic automata without changing the language recognised.

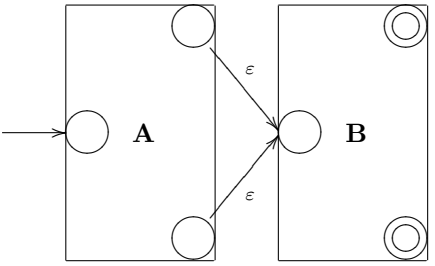
Theorem 2. *Let* $\mathbf{A}$ *be a finite* ε*-acceptor. Then there is an algorithm that constructs a non-deterministic acceptor without* ε*-transitions,* $\mathbf{A}^s$*, such that* $L(\mathbf{A}^s) = L(\mathbf{A})$.

Example 6. We can use ε-acceptors to prove that if L and M are recognisable languages, then so is LM. By assumption, we are given two acceptors $\mathbf{A}$ and $\mathbf{B}$ such that $L(\mathbf{A}) = L$ and $L(\mathbf{B}) = M$. We picture $\mathbf{A}$ and $\mathbf{B}$ schematically as follows:



Now construct the following ε-acceptor: from each terminal state of $\mathbf{A}$ draw an ε-transition to the initial state of $\mathbf{B}$. Make each of the terminal states of

A ordinary states and make the initial state of **B** an ordinary state. Call the resulting acceptor **C**. This can be pictured as follows:



It is easy to see that this ε-acceptor recognises LM. By Theorem 2, we can construct a non-deterministic acceptor recognising LM, and by Theorem 1 we can convert this non-deterministic acceptor into a deterministic acceptor recognising LM. We have therefore proved that LM is recognisable.

Using the idea of Example 6, the following can easily be proved.

Theorem 3. *Let A be an alphabet and L and M be languages over A.*

(i) *If L and M are recognisable then $L + M$ is recognisable.*
(ii) *If L and M are recognisable then LM is recognisable.*
(iii) *If L is recognisable then L^* is recognisable.*

It is worth mentioning that the recognisable languages are closed under all the Boolean operations: thus if L and M are recognisable so too are $L \cap M$, $L + M$, and L'. Furthermore, given finite deterministic acceptors for L and M, it is easy to construct directly finite deterministic acceptors for $L \cap M$, $L + M$, and L'. The proof of this can be found in Chapter 2 of [22].

The explicit algorithms for constructing deterministic acceptors from non-deterministic ones ('the subset construction'), and non-deterministic acceptors from ε-acceptors are described in most books on automata theory; see [16], and Chapters 3 and 4 of [22], for example.

2.4 Kleene's theorem

This is now a good opportunity to reflect on which languages we can now prove are recognisable. I want to pick out four main results. Let $A = \{a_1, \ldots, a_n\}$. Then from Example 4 and Theorem 3, we have the following:

- Each of the languages $\emptyset$, $\{\varepsilon\}$, and $\{a_i\}$ is recognisable.
- The union of two recognisable languages is recognisable.
- The product of two recognisable languages is recognisable.
- The Kleene star of a recognisable language is recognisable.

Call a language over an alphabet *basic* if it is either empty, consists of the empty string alone, or consists of a single symbol from the alphabet. Then what we have proved is the following: a language that can be constructed from the basic languages by using only the operations $+$, $\cdot$, and $*$ a finite number of times must be recognisable. Such a language is said to be *regular* or *rational*.

Example 7. Consider the language L over the alphabet $\{a, b\}$ that consists of all strings of even length. We shall show that this is a regular language. A string of even length is either empty, or can be written as a product of strings of length 2. Conversely every string that can be written as a product of strings of length 2 has even length. It follows that

$$L = ((\{a\}\{a\} + \{a\}\{b\}) + \{b\}\{a\}) + \{b\}\{b\})^*.$$

Thus L is regular.

In the example above, we would much rather write

$$L = (aa + ab + ba + bb)^*$$

for clarity. How to do this in general is formalised in the notion of a 'regular expression.' Let $A = \{a_1, \ldots, a_n\}$ be an alphabet. A *regular expression (over A)* or *rational expression (over A)* is a sequence of symbols formed by repeated application of the following rules:

(R1) $\emptyset$ is a regular expression.
(R2) ε is a regular expression.
(R3) $a_1, \ldots, a_n$ are each regular expressions.
(R4) If s and t are regular expressions then so is $(s + t)$.
(R5) If s and t are regular expressions then so is $(s \cdot t)$.
(R6) If s is a regular expression then so is (s^*).
(R7) Every regular expression arises by a finite number of applications of the rules (R1) through (R6).

As usual, we will write st rather than $s \cdot t$. Each regular expression s describes a regular language, denoted by $L(s)$. This language is calculated by means of the following rules. Simply put, they tell us how to 'insert the curly brackets.'

(D1) $L(\emptyset) = \emptyset$.
(D2) $L(\varepsilon) = \{\varepsilon\}$.
(D3) $L(a_i) = \{a_i\}$.
(D4) $L(s + t) = L(s) + L(t)$.
(D5) $L(s \cdot t) = L(s) \cdot L(t)$.
(D6) $L(s^*) = L(s)^*$.

It is also possible to get rid of many of the left and right brackets that occur in a regular expression by making conventions about the precedence of the regular operators. When this is done, regular expressions form a useful notation for

describing regular languages. However, if a language is described in some other way, it may be necessary to carry out some work to find a regular expression that describes it; Example 7 illustrates this point.

The first major result in automata theory is the following, known as Kleene's theorem.

Theorem 4. *A language is recognisable if and only if it is regular. In particular, there are algorithms that accept as input a regular expression r, and output a finite acceptor $\mathbf{A}$ such that $L(\mathbf{A}) = L(r)$; and there are algorithms that accept as input a finite acceptor $\mathbf{A}$, and output a regular expression r such that $L(r) = L(\mathbf{A})$.*

This theorem is significant for two reasons: first, it tells us that there is an algorithm that enables us to construct an acceptor recognising a language from a suitable description of that language; second, it tells us that there is an algorithm that will produce a description of the language recognised by an acceptor.

A number of different proofs of Kleene's theorem may be found in Chapter 5 of [22]. For further references on how to convert regular expressions into finite acceptors, see [5] and [7]. Regular expressions as I have defined them are useful for proving Kleene's theorem but hardly provide a convenient tool for describing regular languages over realistic alphabets containing large numbers of symbols. The practical side of regular expressions is described by Friedl [14] who shows how to use regular expressions to search texts.

2.5 Minimal automata

In this section, I shall describe an important feature of finite acceptors that makes them particularly useful in applications: the fact that they can be minimised. I have chosen to take the simplest approach in describing this property, but at the end of this section, I describe a more sophisticated one needed in generalisations.

Given a recognisable language L, there will be many finite acceptors that recognise L. All things being equal, we would usually want to pick the smallest such acceptor: namely, one having the smallest number of states. It is conceivable that there could be two different acceptors $\mathbf{A}_1$ and $\mathbf{A}_2$ both recognising L, both having the same number of states, and sharing the additional property that there is no acceptor with fewer states recognising L. In this section, I shall explain why this cannot happen. This result has an important consequence: every recognisable language is accepted by an essentially unique smallest acceptor. To show that this is true, we begin by showing how an acceptor may be reduced in size without changing the language recognised. There are two methods that can be applied, each dealing with a different kind of inefficiency.

The first method removes states that cannot play any role in deciding whether a string is accepted. Let $\mathbf{A} = (S, A, i, \delta, T)$ be a finite acceptor. We

say that a state $s \in S$ is *accessible* if there is a string $x \in A^*$ such that $i \cdot x = s$. Observe that the initial state itself is always accessible because $i \cdot \varepsilon = i$. A state that is not accessible is said to be *inaccessible*. An acceptor is said to be *accessible* if every state is accessible. It is clear that the inaccessible states of an automaton can play no role in accepting strings; consequently, we expect that they could be removed without the language being changed. This turns out to be the case.

Theorem 5. *Let* $\mathbf{A}$ *be a finite acceptor. Then there is an algorithm that constructs an accessible acceptor,* $\mathbf{A}^a$*, such that* $L(\mathbf{A}^a) = L(\mathbf{A})$*.*

The second method identifies states that 'do the same job.' Let $\mathbf{A} = (S, A, i, \delta, T)$ be an acceptor. Two states $s, t \in S$ are said to be *distinguishable* if there exists $x \in A^*$ such that

$$(s \cdot x, t \cdot x) \in (T \times T') \cup (T' \times T),$$

where T' is the set of non-terminal states. In other words, for some string x, one of the states $s \cdot x$ and $t \cdot x$ is terminal and the other non-terminal. The states s and t are said to be *indistinguishable* if they are not distinguishable. This means that for each $x \in A^*$ we have that

$$s \cdot x \in T \Leftrightarrow t \cdot x \in T.$$

Define the relation $\simeq_{\mathbf{A}}$ on the set of states S by

$$s \simeq_{\mathbf{A}} t \Leftrightarrow \text{ } s \text{ and } t \text{ are indistinguishable.}$$

We call $\simeq_{\mathbf{A}}$ the *indistinguishability relation*, and it is an equivalence relation. It can happen, of course, that each pair of states in an acceptor is distinguishable: in other words, the relation $\simeq_{\mathbf{A}}$ is equality. We say that such an acceptor is *reduced*.

Theorem 6. *Let* $\mathbf{A}$ *be a finite acceptor. Then there is an algorithm that constructs a reduced acceptor,* $\mathbf{A}^r$*, such that* $L(\mathbf{A}^r) = L(\mathbf{A})$*.*

Our two methods can be applied to an acceptor $\mathbf{A}$ in turn yielding an acceptor $\mathbf{A}^{ar} = (\mathbf{A}^a)^r$ that is both accessible and reduced. The reader may wonder at this point whether there are other methods for removing states. We shall see that there are not.

We now come to a fundamental definition. Let L be a recognisable language. A finite deterministic acceptor $\mathbf{A}$ is said to be *minimal (for L)* if $L(\mathbf{A}) = L$, and if $\mathbf{B}$ is any finite acceptor such that $L(\mathbf{B}) = L$, then the number of states of $\mathbf{A}$ is less than or equal to the number of states of $\mathbf{B}$. Minimal acceptors for a language L certainly exist. The problem is to find a way of constructing them. Our search is narrowed down by the following observation whose simple proof is left as an exercise: if $\mathbf{A}$ is minimal for L, then $\mathbf{A}$ is both accessible and reduced.

In order to realise the main goal of this section, we need to have a precise mathematical definition of when two acceptors are essentially the same: one that we can check in a systematic way however large the automata involved. The definition below provides the answer to this question.

Let $\mathbf{A} = (S, A, s_0, \delta, F)$ and $\mathbf{B} = (Q, A, q_0, \gamma, G)$ be two acceptors with the same input alphabet A. An *isomorphism* θ *from* $\mathbf{A}$ *to* $\mathbf{B}$ is a function $\theta\colon\ S \to Q$ satisfying the following four conditions:

(IM1) The function θ is bijective.
(IM2) $\theta(s_0) = q_0$.
(IM3) $s \in F \Leftrightarrow \theta(s) \in G$.
(IM4) $\theta(\delta(s, a)) = \gamma(\theta(s), a)$ for each $s \in S$ and $a \in A$.

If there is an isomorphism from $\mathbf{A}$ to $\mathbf{B}$ we say that $\mathbf{A}$ is *isomorphic* to $\mathbf{B}$. Isomorphic acceptors may differ in their state labeling and may look different when drawn as directed graphs, but by suitable relabeling, and by moving states and bending transitions, they can be made to look identical. Thus isomorphic automata are 'essentially the same' meaning that they differ in only trivial ways.

Theorem 7. *Let L be a recognisable language. Then L has a minimal acceptor, and any two minimal acceptors for L are isomorphic. A reduced accessible acceptor recognising L is a minimal acceptor for L.*

Remark It is worth reflecting on the significance of this theorem, particularly since in the generalisations considered later in this chapter, a rather more subtle notion of 'minimal automaton' has to be used. Theorem 7 tells us that if by some means we can find an acceptor for a language, then by applying a couple of algorithms, we can convert it into the smallest possible acceptor for that language. This should be contrasted with the situation for arbitrary problems where, if we find a solution, there are no general methods for making it more efficient, and where the concept of a smallest solution does not even make sense. The above theorem is therefore one of the benefits of working with a restricted class of operations.

The approach I have adopted to describing a minimal acceptor can be generalised in a straightforward fashion to the Moore and Mealy machines I describe in Section 3.1. However, when I come to the sequential transducers of Section 3.2, this naive approach breaks down. In this case, it is indeed possible to have two sequential transducers that do the same job, are as small as possible, but which are not isomorphic. A specific example of this phenomenon can be found in [27]. However, it transpires that we can still pick out a 'canonical machine' that also has the smallest possible number of states. The construction of this canonical machine needs slightly more sophisticated mathematics; I shall outline how this approach can be carried out for finite acceptors.

The finite acceptors I have defined are technically speaking the 'complete finite acceptors.' An *incomplete* finite acceptor is defined in the same way as a complete one except that we allow the transition function to be a partial

function. Clearly, we can convert an incomplete acceptor into a complete one by adjoining an extra state and defining appropriate transitions. However, there is no need to do this: incomplete acceptors bear the same relationship to complete ones as partial functions do to (globally defined) functions, and in computer science it is the partial functions that are the natural functions to consider. For the rest of this paragraph, 'acceptor' will mean one that could be incomplete. One way of simplifying an acceptor is to remove the inaccessible states. Another way of simplifying an acceptor is to remove those states s for which there is no string x such that $s \cdot x$ is terminal. An acceptor with the property that for each state s there is a string x such that $s \cdot x$ is terminal is said to be *coaccessible*. Clearly, if we prune an acceptor of those states that are not coaccessible, the resulting acceptor is coaccessible. The reader should observe that if this procedure is carried out on a complete acceptor, then the resulting acceptor could well be incomplete. This is why I did not define this notion earlier. Acceptors that are both accessible and coaccessible are said to be *trim*. It is possible to define what we mean by a 'morphism' between acceptors; I shall not make a formal definition here, but I will explain how they can be used. Let L be a recognisable language, and consider all the trim acceptors recognising L. If $\mathbf{A}$ and $\mathbf{B}$ are two such acceptors, it can be proved that there is *at most one* morphism from $\mathbf{A}$ to $\mathbf{B}$. If there is a morphism from $\mathbf{A}$ to $\mathbf{B}$, and from $\mathbf{B}$ to $\mathbf{A}$, then $\mathbf{A}$ and $\mathbf{B}$ are said to be 'isomorphic'; this has the same significance as my earlier definition of isomorphic. The key point now is this:

> *there is a distinguished trim acceptor* $\mathbf{A}_L$ *recognising* L *characterised by the property that for each trim acceptor* $\mathbf{A}$ *recognising* L *there is a, necessarily unique, morphism from* $\mathbf{A}$ *to* $\mathbf{A}_L$.

It turns out that $\mathbf{A}_L$ can be obtained from $\mathbf{A}$ by a slight generalisation of the reduction process I described earlier. By definition, $\mathbf{A}_L$ is called the *minimal acceptor for* L. It is a consequence of the defining property of $\mathbf{A}_L$ that $\mathbf{A}_L$ has the smallest number of states amongst all the trim acceptors recognising L. The reader may feel that this description of the minimal acceptor merely complicates my earlier, more straightforward, description. However, the important point is this: the characterisation of the minimal acceptor in the terms I have highlighted above generalises, whereas its characterisation in terms of having the smallest possible number of states does not. A full mathematical justification of the claims made in this paragraph can be found in Chapter III of [12].

A simple algorithm for minimising an acceptor and an algorithm for constructing a minimal acceptor from a regular expression are described in Chapter 7 of [22]. For an introduction to implementing finite acceptors and their associated algorithms, see [34].

3 Finite Transducers

In Section 2, I outlined the theory of finite acceptors. This theory tells us about devices where the response to an input is simply a 'yes' or a 'no'. In this section, I describe devices that generalise acceptors but generate outputs that provide more illuminating answers to questions. Section 3.1 describes how to modify acceptors so that they generate output. It turns out that there are two ways to do this: either to associate outputs with states, or to associate outputs with transitions. The latter approach is the one adopted for generalisations. Section 3.2 describes the most general way of generating output in a sequential fashion, and Section 3.3 describes the most general model of 'finite state devices.'

3.1 Finite purely sequential transducers

I shall begin this section by explaining how finite acceptors can be adapted to generate outputs.

A language L is defined to be a subset of some A^*, where A is any alphabet. Subsets of sets can also be defined in terms of functions. To see how, let X be a set, and let $Y \subseteq X$ be any subset. Define a function

$$\chi_Y \colon\ X \to \mathbf{2} = \{0, 1\}$$

by

$$\chi_Y(x) = \begin{cases} 1 \text{ if } x \in Y \\ 0 \text{ if } x \notin Y. \end{cases}$$

The function χ_Y, which contains all the information about which elements belong to the subset Y, is called the *characteristic function* of the subset Y. More generally, *any* function

$$\chi \colon\ X \to \mathbf{2}$$

defines a subset of X: namely, the set of all $x \in X$ such that $\chi(x) = 1$. It is not hard to see that subsets of X and characteristic functions on X are equivalent ways of describing the same thing. It follows that languages over A can be described by functions $\chi \colon\ A^* \to \mathbf{2}$, and vice versa.

Suppose now that L is a language recognised by the acceptor $\mathbf{A} = (Q, A, i, \delta, T)$. We should like to regard $\mathbf{A}$ as calculating the characteristic function χ_L of L. To do this, we need to make some minor alterations to $\mathbf{A}$. Rather than labeling a state as terminal, we shall instead add the label '1' to the state; thus if the state q is terminal, we shall relabel it as $q/1$. If a state q is not terminal, we shall relabel it as $q/0$. Clearly with this labeling, we can dispense with the set T since it can be recovered as those states q labeled '1'. What we have done is define a function $\lambda \colon\ Q \to \mathbf{2}$. Our 'automaton' is now described by the following information: $\mathbf{B} = (Q, A, \mathbf{2}, q_0, \delta, \lambda)$. To see how this

automaton computes the characteristic function, we need an auxiliary function $\omega_{\mathbf{B}}\colon A^* \to (0+1)^*$, which is defined as follows. Let $x = x_1 \dots x_n$ be a string of length n over A, and let the states $\mathbf{B}$ passes through when processing x be $q_0, q_1, \dots, q_n$. Thus

$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \dots \xrightarrow{x_n} q_n.$$

Define the string

$$\omega_{\mathbf{B}}(x) = \lambda(q_0)\lambda(q_1)\dots\lambda(q_n).$$

Thus $\omega_{\mathbf{B}}\colon A^* \to (0+1)^*$ is a function such that

$$\omega_{\mathbf{B}}(\varepsilon) = \lambda(q_0) \text{ and } |\omega_{\mathbf{B}}(x)| = |x| + 1.$$

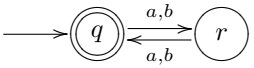
The characteristic function of the language $L(\mathbf{A})$ is the function $\rho\omega_{\mathbf{B}}\colon A^* \to \mathbf{2}$, where ρ is the function that outputs the rightmost letter of a non-empty string.

For the automaton $\mathbf{B}$, I have defined two functions: $\omega_{\mathbf{B}}\colon A^* \to (0+1)^*$, which I shall call the *output response function* of the automaton $\mathbf{B}$, and $\chi_{\mathbf{B}}\colon A^* \to \mathbf{2}$, which I shall call the *characteristic function* of the automaton $\mathbf{B}$. I shall usually just write ω and χ when the automaton in question is clear. We have noted already that $\chi = \rho\omega$. On the other hand,

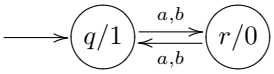
$$\omega(x_1 \dots x_n) = \chi(\varepsilon)\chi(x_1)\chi(x_1x_2)\dots\chi(x_1\dots x_n).$$

Thus knowledge of either one of ω and χ is enough to determine the other; both are legitimate output functions, and which one we use will be decided by the applications we have in mind. To make these ideas more concrete, here is an example.

Example 8. Consider the finite acceptor $\mathbf{A}$ below



The language $L(\mathbf{A})$ consists of all those strings in $(a+b)^*$ of even length. We now convert it into the automaton $\mathbf{B}$ described above



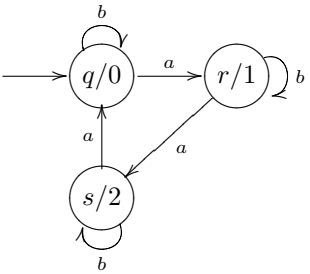
We can calculate the value of the output response function $\omega\colon (a+b)^* \to (0+1)^*$ on the string aba by observing that in processing this string we pass through the four states: q, r, q, and r. Thus $\omega(aba) = 1010$. By definition, $\chi(aba) = 0$.

There is nothing sacrosanct about the set $\mathbf{2}$ having two elements. We could just as well replace it by any alphabet B, and so view an automaton as

computing functions from A^* to B. This way of generating output from an automaton leads to the following definition.

A finite *Moore machine*, $\mathbf{A} = (Q, A, B, q_0, \delta, \lambda)$, consists of the following ingredients: Q is a finite set of states, A is the *input alphabet*, B is the *output alphabet*, q_0 is the initial state, $\delta\colon Q \times A \to Q$ is the transition function, and $\lambda\colon Q \to B$ tells us the output associated with each state. As in our special case where $B = \mathbf{2}$, we can define the *output response function* $\omega_{\mathbf{A}}\colon A^* \to B^*$ and the *characteristic function* $\chi_{\mathbf{A}}\colon A^* \to B$; as before, knowing one of these functions means we know the other. When drawing transition diagrams of Moore machines the function λ is specified on the state q by labeling this state $q/\lambda(q)$. The same idea can be used if the Moore machine is specified by a transition table.

Example 9. Here is an example of a finite Moore machine where the output alphabet has more than two letters.



Here the input alphabet $A = \{a, b\}$ and the output alphabet is $B = \{0, 1, 2\}$. In the table below, I have calculated the values of $\omega(x)$ and $\chi(x)$ for various input strings x.

x	$\omega(x)$	$\chi(x)$
ε	0	0
a	01	1
b	00	0
aa	012	2
ab	011	1
ba	001	1
bb	000	0
aaa	0120	0
aab	0122	2
aba	0112	2
abb	0111	1
baa	0012	2
bab	0011	1
bba	0001	1
bbb	0000	0

It is natural to ask under what circumstances a function $f\colon A^* \to B$ is the characteristic function of some finite Moore machine. One answer to this question is provided by the theorem below, which can be viewed as an application of Kleene's theorem. For a proof see Theorem XI.6.1 of [12].

Theorem 8. *Let $f\colon A^* \to B$ be an arbitrary function. Then there is a finite Moore machine $\mathbf{A}$ with input alphabet A and output alphabet B such that $f = \chi_{\mathbf{A}}$, the characteristic function of $\mathbf{A}$, if and only if for each $b \in B$ the language $f^{-1}(b)$ is regular.*

Moore machines are not the only way in which output can be generated. A *Mealy machine* $\mathbf{A} = (Q, A, B, q_0, \delta, \lambda)$ consists of the following ingredients: Q is a finite set of states, A is the input alphabet, B is the output alphabet, q_0 is the initial state, $\delta\colon Q \times A \to Q$ is the transition function, and $\lambda\colon Q \times A \to B$ associates an output symbol with each transition. The *output response function* $\omega_{\mathbf{A}}\colon A^* \to B^*$ of the Mealy machine $\mathbf{A}$ is defined as follows. Let $x = x_1 \dots x_n$ be a string of length n over A, and let the states $\mathbf{A}$ passes through when processing x be $q_0, q_1, \dots, q_n$. Thus

$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \dots \xrightarrow{x_n} q_n.$$

Define

$$\omega_{\mathbf{A}}(x) = \lambda(q_0, x_1)\lambda(q_1, x_2) \dots \lambda(q_{n-1}, x_n).$$

Thus $\omega_{\mathbf{A}}\colon A^* \to (0+1)^*$ is a function such that

$$\omega_{\mathbf{A}}(\varepsilon) = \varepsilon \text{ and } |\omega_{\mathbf{A}}(x)| = |x|.$$

Although Moore machines generate output when a state is entered, and Mealy machines during a transition, the two formalisms have essentially the same power. The simple proofs of the following two results can be found as Theorems 2.6 and 2.7 of [16].

Theorem 9. *Let A and B be finite alphabets.*

(i) *Let $\mathbf{A}$ be a finite Moore machine with input alphabet A and output alphabet B. Then there is a finite Mealy machine $\mathbf{B}$ with the same input and output alphabets and a symbol $a \in A$ such that $\chi_{\mathbf{A}} = a\chi_{\mathbf{B}}$.*

(ii) *Let $\mathbf{A}$ be a finite Mealy machine with input alphabet A and output alphabet B. Then there is a finite Moore machine $\mathbf{B}$ with the same input and output alphabets and a symbol $a \in A$ such that $\chi_{\mathbf{B}} = a\chi_{\mathbf{A}}$.*

A partial function $f\colon A^* \to B^*$ is said to be *prefix-preserving* if for all $x, y \in A^*$ such that $f(xy)$ is defined, the string $f(x)$ is a prefix of $f(xy)$. From Theorems 8 and 9, we may deduce the following characterisation of the output response functions of finite Mealy machines.

Theorem 10. *A function $f\colon A^* \to B^*$ is the output response function of a finite Mealy machine if and only if the following three conditions hold:*

(i) $|f(x)| = |x|$ *for each* $x \in A^*$.
(ii) f *is prefix-preserving.*
(iii) *The set* $f^{-1}(X)$ *is a regular subset of* A^* *for each regular subset* $X \subseteq B^*$.

Both finite Moore machines and finite Mealy machines can be minimised in a way that directly generalises the minimisation of automata described in Section 2.5. The details can be found in Chapter 7 of [9], for example.

Mealy machines provide the most convenient starting point for the further development of the theory of finite automata, so for the remainder of this section I shall concentrate solely on these. There are two ways in which the definition of a finite Mealy machine can be generalised. The first is to allow both δ, the transition function, and λ, the output associated with a transition, to be *partial* functions. This leads to what are termed *incomplete finite Mealy machines*. The second is to define $\lambda\colon\ Q \times A \to B^*$; in other words, we allow an input *symbol* to give rise to an output *string*. If both these generalisations are combined, we get the following definition.

A *finite (left) purely sequential transducer* $\mathbf{A} = (Q, A, B, q_0, \delta, \lambda)$ consists of the following ingredients: Q is a finite set of states, A is the input alphabet, B is the output alphabet, q_0 is the initial state, $\delta\colon Q \times A \to Q$ is a partial function, called the transition function, and $\lambda\colon\ Q \times A \to B^*$ is a partial function that associates an output string with each transition. The *output response function* $\omega_{\mathbf{A}}\colon\ A^* \to B^*$ of the purely sequential transducer $\mathbf{A}$ is a partial function defined as follows. Let $x = x_1 \dots x_n$ be a string of length n over A, and suppose that x labels a path in $\mathbf{A}$ that starts at the initial state; thus

$$q_0 \xrightarrow{x_1} q_1 \xrightarrow{x_2} \dots \xrightarrow{x_n} q_n.$$

Define the string

$$\omega_{\mathbf{A}}(x) = \lambda(q_0, x_1)\lambda(q_1, x_2) \dots \lambda(q_{n-1}, x_n).$$

I have put the word 'left' in brackets; it refers to the fact that in the definition of δ and λ we read the input string from left to right. If instead we read the input string from right to left, we would have what is known as a *finite right purely sequential transducer*. I shall assume that a finite purely sequential transducer is a left one unless otherwise stated. A partial function $f\colon\ A^* \to B^*$ is said to be *(left) purely sequential* if it is the output response function of some finite (left) purely sequential transducer. *Right purely sequential* partial functions are defined analogously. The notions of left and right purely sequential functions are distinct, and there are partial functions that are neither.

The following theorem generalises Theorem 10 and was first proved in [15]. A proof can be found in [4] as Theorem IV.2.8.

Theorem 11. *A partial function* $f\colon\ A^* \to B^*$ *is purely sequential if and only if the following three conditions hold:*

(i) *There is a natural number n such that if x is a string in A^*, and $a \in A$, and $f(xa)$ is defined, then*

$$|f(xa)| - |f(x)| \leq n.$$

(ii) *f is prefix-preserving.*
(iii) *The set $f^{-1}(X)$ is a regular subset of A^* for each regular subset $X \subseteq B^*$.*

The theory of minimising finite acceptors can be extended to finite purely sequential transducers. See Chapter XII, Section 4 of [12].

The final result of this section is proved as Proposition IV.2.5 of [4].

Theorem 12. *Let $f\colon A^* \to B^*$ and $g\colon B^* \to C^*$ be left (resp. right) purely sequential partial functions. Then their composition $g \circ f\colon A^* \to C^*$ is a left (resp. right) purely sequential partial function.*

3.2 Finite sequential transducers

The theories of recognisable languages and purely sequential partial functions outlined in Sections 2 and 3.1 can be regarded as the classical theory of finite automata. For example, the Mealy and Moore machines discussed in Section 3.1, particularly in their incomplete incarnations, form the theoretical basis for the design of circuits. But although purely sequential functions are useful, they have their limitations. For example, binary addition cannot quite be performed by means of a finite purely sequential transducer (see Example IV.2.4 and Exercise IV.2.1 of [4]). This led Schützenberger [33] to introduce the 'finite sequential transducers' and the corresponding class of 'sequential partial functions.' The definition of a finite sequential transducer looks like a cross between finite acceptors and finite purely sequential transducers.

A *finite sequential transducer*, $\mathbf{A} = (Q, A, B, q_0, \delta, \lambda, \tau, x_i)$, consists of the following ingredients: Q is a finite set of states, A is the input alphabet, B is the output alphabet, q_0 is the initial state, $\delta\colon Q \times A \to Q$ is a transition partial function, $\lambda\colon Q \times A \to B^*$ is an output partial function, $\tau\colon T \to B^*$ is the *termination* function, where T is a subset of Q called the set of terminal states, and $x_i \in B^*$ is the *initialisation value*.

To see how this works, let $x = x_1 \dots x_n$ be an input string from A^*. We say that x is *successful* if it labels a path from q_0 to a state in T. For those strings $x \in A^*$ that are successful, we define an output string from B^* as follows: the initialisation value x_i is concatenated with the output response string determined by x and the function λ, just as in a finite purely sequential transducer, and then concatenated with the string $\tau(q_0 \cdot x)$. In other words, the output is computed in the same way as in a finite purely sequential transducer except that this is prefixed by a fixed string and suffixed by a final output string determined by the last state. Partial functions from A^* to B^* that can be computed by some finite sequential transducer are called *sequential*

partial functions.[1] Finite sequential transducers can be represented by suitably modified transition diagrams: the initial state is represented by an inward-pointing arrow labeled by x_i, and each terminal state t is marked by an outward-pointing arrow labeled by $\tau(t)$.

Every purely sequential function is sequential, but there are sequential functions that are not purely sequential. Just as with purely sequential transducers, finite sequential transducers can be minimised, although the procedure is necessarily more complex; see [27] and [11] for details and the Remark at the end of Section 2.5; in addition, the composition of sequential functions is sequential. A good introduction to sequential partial functions and to some of their applications is the work in [27].

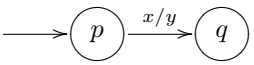
3.3 Finite transducers

In this section, we arrive at our final class of automata, which contains all the automata we have discussed so far as special cases.

A *finite transducer*, $\mathbf{T} = (Q, A, B, q_0, E, F)$, consists of the following ingredients: a finite set of states Q, an input alphabet A, an output alphabet B, an initial state q_0,[2] a set of final or terminal states F, and a set E of transitions where

$$E \subseteq Q \times A^* \times B^* \times Q.$$

A finite transducer can be represented by means of a transition diagram where each transition has the form



where $(p, x, y, q) \in E$. As usual, we indicate the initial state by an inward-pointing arrow and the final states by double circles.

To describe what a finite transducer does, we need to introduce some notation. Let

$$e = (q_1, x_1, y_1, q_1') \dots (q_n, x_n, y_n, q_n')$$

be any sequence of transitions. The state q_1 will be called the *beginning of* e and the state q_n' will be called the *end of* e. The *label* of e is the pair of strings

$$(x_1 \dots x_n, y_1 \dots y_n).$$

If e is the empty string then its label is $(\varepsilon, \varepsilon)$. We say that a sequence of transitions e is *allowable* if it describes an actual path in $\mathbf{T}$; this simply means that for each consecutive pair

$$(q_i, x_i, y_i, q_i')(q_{i+1}, x_{i+1}, y_{i+1}, q_{i+1}')$$

[1] Berstel [4] does not include in his definition the string x_i (alternatively, he assumes that $x_i = \varepsilon$). However, the class of partial functions defined is the same.

[2] Sometimes a *set* of initial states is allowed; this does not change the theory.

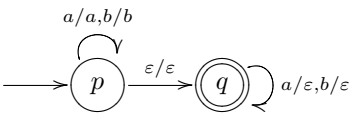
in e we have that $q'_i = q_{i+1}$. An allowable sequence e is said to be *successful* if it begins at the initial state and ends at a terminal state. Define the relation

$$R(\mathbf{T}) = \{(x, y) \in A^* \times B^* \colon\ (x, y) \text{ is the label of a successful path in } \mathbf{T}\}.$$

We call $R(\mathbf{T})$ the *relation computed by* $\mathbf{T}$.

Observe that in determining the $y \in B^*$ such that $(x, y) \in R(\mathbf{T})$ for a given x, the transducer $\mathbf{T}$ processes the string x in the manner of an ε-acceptor. Thus we need to search for those paths in $\mathbf{T}$ starting at the initial state and ending at a terminal state such that the sequence of labels $(a_1, b_1), \ldots, (a_n, b_n)$ encountered has the property that the concatenation $a_1 \ldots a_n$ is equal to x where some of the a_i may well be ε.

Example 10. The following is a transition diagram of a transducer $\mathbf{T}$



In this case, the input and output alphabets are the same and equal $\{a, b\}$. The relation $R(\mathbf{T}) \subseteq (a + b)^* \times (a + b)^*$ computed by $\mathbf{T}$ is the set of all pairs of strings (x, y) such that y is a prefix of x. This is a relation rather than a function because a non-empty string has more than one prefix.

The relations in $A^* \times B^*$ that can be computed by finite transducers can be described in a way that generalises Kleene's theorem. To see how, we need to define what we mean by a 'regular' or 'rational' subset of $A^* \times B^*$. If $(x_1, y_1), (x_2, y_2) \in A^* \times B^*$, then we define their *product* by

$$(x_1, y_1)(x_2, y_2) = (x_1 x_2, y_1 y_2).$$

This operation is the analogue of concatenation in A^*. Observe that $(\varepsilon, \varepsilon)$ has the property that $(\varepsilon, \varepsilon)(x, y) = (x, y) = (x, y)(\varepsilon, \varepsilon)$. If $L, M \subseteq A^* \times B^*$, then define LM to be the set of all products of elements in L followed by elements in M. With these preliminaries out of the way, we can define a *regular* or *rational* subset of $A^* \times B^*$ in a way analogous to the definition of a regular subset given in Section 2.4. A regular subset of $A^* \times B^*$ is also called a *regular* or *rational relation* from A^* to B^*.

The following result is another application of Kleene's Theorem; see Theorem III.6.1 of [4] for a proof and [29] for the correct mathematical perspective on finite transducers.

Theorem 13. *A relation $R \subseteq A^* \times B^*$ can be computed by a finite transducer with input alphabet A and output alphabet B if and only if R is a regular relation from A^* to B^*.*

In the Introduction, I indicated that there were simple computations that finite transducers could not do. A good example is that of reversing a string; see Exercise III.3.2 of [4].

The theory of finite transducers is more complex than that of finite acceptors. In what follows, I just touch on some of the key points.

The following is proved as Theorem III.4.4 of [4].

Theorem 14. *Let A, B, C be finite alphabets. Let R be a regular relation from A^* to B^* and let R' be a regular relation from B^* to C^*. Then $R' \circ R$ is a regular relation from A^* to C^*, where $(a, c) \in R' \circ R$ iff there exists $b \in B^*$ such that $(a, b) \in R'$ and $(b, c) \in R$.*

Let R be a regular relation from A^* to B^*. Given a string $x \in A^*$, there may be no strings y such that $(x, y) \in R$; there might be exactly one such string y; or they might be many such strings y. If a relation R from A^* to B^* has the property that for each $x \in A^*$ there is at most one element $y \in B^*$ such that $(x, y) \in R$, then R can be regarded as a partial function from A^* to B^*. Such a function is called a *regular* or *rational partial function*.

It is important to remember that a regular relation that is not a regular partial function is not, in some sense, deficient; there are many situations where it would be unrealistic to expect a partial function. For example, in natural language processing, regular relations that are not partial functions can be used to model ambiguity of various kinds. However, regular partial functions are easier to handle. For example, there is an algorithm that will determine whether two regular partial functions are equal or not (Corollary IV.1.3 [4]), whereas there is no such algorithm for arbitrary regular relations (Theorem III.8.4(iii) [4]). Classes of regular relations sharing some of the advantages of sequential partial functions are described in [1] and [23].

Theorem 15. *There is an algorithm that will determine whether the relation computed by a finite transducer is a partial function or not.*

This was first proved by Schützenberger [32], and a more recent paper [3] also discusses this question.

Both left and right purely sequential functions are examples of regular partial functions, and there is an interesting relationship between arbitrary regular partial functions and the left and right purely sequential ones. The following is proved as Theorem IV.5.2 of [4].

Theorem 16. *Let $f\colon A^* \to B^*$ be a partial function such that $f(\varepsilon) = \varepsilon$. Then f is regular if and only if there is an alphabet C and a left purely sequential partial function $f_L\colon A^* \to C^*$ and a right purely sequential partial function $f_R\colon C^* \to B^*$ such that $f = f_R \circ f_L$.*

The idea behind the theorem is that to compute $f(x)$ we can first process x from left to right and then from right to left. Minimisation of machines that

compute regular partial functions is more complex and less clear-cut than for the sequential ones; see [28] for some work in this direction.

The sequential functions are also regular partial functions. The following definition is needed to characterise them. Let x and y be two strings over the same alphabet. We denote by $x \wedge y$ the *longest common prefix of x and y*. We define the *prefix distance* between x and y by

$$d(x, y) = |x| + |y| - 2|x \wedge y|.$$

In other words, if $x = ux'$ and $y = uy'$, where $u = x \wedge y$, then $d(x, y) = |x'| + |y'|$. A partial function $f\colon A^* \to B^*$ is said to have *bounded variation* if for each natural number n there exists a natural number N such that, for all strings $x, y \in A^*$,

$$\text{if } d(x, y) \leq n \text{ then } d(f(x), f(y)) \leq N.$$

Theorem 17. *Every sequential function is regular. In particular, the sequential functions are precisely the regular partial functions with bounded variation.*

The proof of the second claim in the above theorem was first given by Choffrut [10]. Both proofs can be found in [4] (respectively, Proposition IV.2.4 and Theorem IV.2.7).

Theorem 18. *There is an algorithm that will determine whether a finite transducer computes a sequential function.*

This was first proved by Choffrut [10], and more recent papers that discuss this question are [2] and [3].

Finite transducers were introduced in [13] and, as we have seen, form a general framework containing the purely sequential and sequential transducers.

4 Final Remarks

In this section, I would like to touch on some of the practical reasons for using finite transducers. But first, I need to deal with the obvious objection to using them: that they cannot implement all algorithms, because they do not constitute a universal programming language. However, it is the very lack of ambition of finite transducers that leads to their virtues: we can say more about them, and what we can say can be used to help us design programs using them. The manipulation of programs written in universal programming languages, on the other hand, is far more complex. In addition:

- Not all problems require for their solution the full weight of a universal programming language — if we can solve them using finite transducers then the benefits described below will follow.

- Even if the full solution of a problem does fall outside the scope of finite transducers, the cases of the problem that are of practical interest may well be described by finite transducers. Failing that, partial or approximate solutions that can be described by finite transducers may be acceptable for certain purposes.
- It is one of the goals of science to understand the nature of problems. If a problem can be solved by a finite transducer then we have learnt something non-trivial about the nature of that problem.

The benefits of finite transducers are particularly striking in the case of finite acceptors:

- Finite acceptors provide a way to describe potentially infinite languages in finite ways.
- Determining whether a string is accepted or rejected by a deterministic acceptor is linear in the length of the string.
- Acceptors can be both determinised and minimised.

The important point to bear in mind is that languages are interesting because they can be used to encode structures of many kinds. An example from mathematics may be instructive. A *relational structure* is a set equipped with a finite collection of relations of different arities. For example, a set equipped with a single binary relation is just a graph with at most one edge joining any two vertices. We say that a relational structure is *automatic* if the elements of the set can be encoded by means of strings from a regular language, and if each of the n-ary relations of the structure can be encoded by means of an acceptor. Encoding n-ary relations as languages requires a way of encoding n-tuples of strings as single strings, but this is easy and the details are not important; see [19] for the complete definition and [8] for a technical analysis of automatic structures from a logical point of view. Minimisation means that encoded structures can be compressed with no loss of information. Now there are many ways of compressing data, but acceptors provide an additional advantage: they come with a built-in search capability. The benefits of using finite acceptors generalise readily to finite sequential transducers.

There is a final point that is worth noting. Theorem 14 tells us that composing a sequence of finite transducers results in another finite transducer. This can be turned around and used as a design method; rather than trying to construct a finite transducer in one go, we can try to design it as the composition of a sequence of simpler finite transducers.

The books [18, 20, 31] although dealing with natural language processing contain chapters that may well provide inspiration for applications of finite transducers to other domains.

Acknowledgments

It is a pleasure to thank a number of people who helped in the preparation of this article.

Dr Karin Haenelt of the Fraunhofer Gesellschaft, Darmstadt, provided considerable insight into the applications of finite transducers in natural language processing which contributed in formulating Section 4.

Dr Jean-Eric Pin, director for research of CNRS, made a number of suggestions including the choice of terminology and the need to clarify what is meant by the notion of minimisation.

Dr Anne Heyworth of the University of Leicester made a number of useful textual comments as well as providing the automata diagrams.

Prof John Fountain and Dr Victoria Gould of the University of York made a number of helpful comments on the text.

My thoughts on the role of finite transducers in information processing were concentrated by a report I wrote for DSTL in July 2003 (Contract number RD026-00403) in collaboration with Peter L. Grainger of DSTL.

Any errors remaining are the sole responsibility of the author.

References

1. C. Allauzen and M. Mohri, Finitely subsequential transducers, *International J. Foundations Comp. Sci.* **14** (2003), 983–994.
2. M.-P. Béal and O. Carton, Determinization of transducers over finite and infinite words, *Theoret. Comput. Sci.* **289** (2002), 225–251.
3. M.-P. Béal, O. Carton, C. Prieur, and J. Sakarovitch, Squaring transducers, *Theoret. Comput. Sci.* **292** (2003), 45–63.
4. J. Berstel, *Transductions and Context-Free Languages*, B.G. Teubner, Stuttgart, 1979.
5. J. Berstel and J.-E. Pin, Local languages and the Berry-Sethi algorithm, *Theoret. Comput. Sci.* **155** (1996), 439–446.
6. J. Berstel and D. Perrin, Algorithms on words, in *Applied Combinatorics on Words* (editor M. Lothaire), in preparation, 2004.
7. A. Brüggemann-Klein, Regular expressions into finite automata, *Lecture Notes in Computer Science* **583** (1992), 87–98.
8. A. Blumensath, *Automatic structures*, Diploma Thesis, Rheinisch-Westfälische Technische Hochschule Aachen, Germany, 1999.
9. J. Carroll and D. Long, *Theory of Finite Automata*, Prentice-Hall, Englewood Cliffs, NJ, 1989.
10. Ch. Choffrut, Une caractérisation des fonctions séquentielles et des fonctions sous-séquentielles en tant que relations rationelles, *Theoret. Comput. Sci.* **5** (1977), 325–337.
11. Ch. Choffrut, Minimizing subsequential transducers: a survey, *Theoret. Comput. Sci.* **292** (2003), 131–143.
12. S. Eilenberg, *Automata, Languages, and Machines*, Volume A, Academic Press, New York, 1974.

13. C. C. Elgot and J. E. Mezei, On relations defined by generalized finite automata, *IBM J. Res. Develop.* **9** (1965), 47–65.
14. J. E. F. Friedl, *Mastering regular expressions*, O'Reilly, Sebastopol, CA, 2002.
15. S. Ginsburg and G. F. Rose, A characterization of machine mappings, *Can. J. Math.* **18** (1966), 381–388.
16. J. E. Hopcroft and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.
17. J. E. Hopcroft, R. Motwani, and J. D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 2nd Edition, Addison-Wesley, Reading, MA, 2001.
18. D. Jurafsky and J. H. Martin, *Speech and Language Processing*, Prentice-Hall, Englewood Cliffs, NJ, 2000.
19. B. Khoussainov and A. Nerode, Automatic presentations of structures, *Lecture Notes in Computer Science* **960** (1995), 367–392.
20. A. Kornai (editor), *Extended Finite State Models of Language*, Cambridge University Press, London, 1999.
21. E. Laporte, Symbolic natural language processing, in *Applied Combinatorics on Words* (editor M. Lothaire), in preparation, 2004.
22. M. V. Lawson, *Finite Automata*, CRC Press, Boca Raton, FL, 2003.
23. M. Mohri, Finite-state transducers in language and speech processing, *Comput. Linguistics* **23** (1997), 269–311.
24. M. Pelletier and J. Sakarovitch, On the representation of finite deterministic 2-tape automata, *Theoret. Comput. Sci.* **225** (1999), 1–63.
25. D. Perrin, Finite automata, in *Handbook of Theoretical Computer Science*, Volume B (editor J. Van Leeuwen), Elsevier, Amsterdam, 1990, 1–57.
26. D. Perrin and J. E. Pin, *Infinite Words*, Elsevier, Amsterdam, 2004.
27. Ch. Reutenauer, Subsequential functions: characterizations, minimization, examples, *Lecture Notes in Computer Science* **464** (1990), 62–79.
28. Ch. Reutenauer and M.-P. Schützenberger, Minimization of rational word functions, *Siam. J. Comput.* **20** (1991), 669–685.
29. J. Sakarovitch, Kleene's theorem revisited, *Lecture Notes in Computer Science* **281** (1987), 39–50.
30. J. Sakarovitch, *Eléments de Théorie des Automates*, Vuibert, Paris, 2003.
31. E. Roche and Y. Schabes (editors), *Finite-State Language Processing*, The MIT Press, 1997.
32. M.-P. Schützenberger, Sur les relations rationelles, in *Automata theory and formal languages* (editor H. Brakhage), *Lecture Notes in Computer Science* **33** (1975), 209–213.
33. M.-P. Schützenberger, Sur une variante des fonctions séquentielles, *Theoret. Comput. Sci.* **4** (1977), 47–57.
34. B. W. Watson, Implementing and using finite automata toolkits, in *Extended Finite State Models of Language* (editor A. Kornai), Cambridge University Press, London, 1999, 19–36.
35. S. Yu, Regular languages, in *Handbook of Formal Languages*, Volume 1 (editors G. Rozenberg and A. Salomaa), Springer-Verlag, Berlin, 1997, 41–110.

Basics of Computer Architecture

Charles B. Silio, Jr.

Electrical and Computer Engineering Department
University of Maryland
College Park, MD 20742-3285, U.S.A.
silio@eng.umd.edu

1 Introduction

User-programmable electronic digital computers have progressed through a series of hardware implementations. They began first as cabinets filled with vacuum tubes requiring steel I-beams to support their weight (hence the term "mainframe" to refer to the central processing unit), then as cabinets filled with discrete transistor circuits on cards, then progressed to smaller implementations known as minicomputers (some capable of residing on a desktop), and finally to microprocessors with the components of complete computers residing on only a few integrated circuit chips or even a complete system on only one chip. The term microprocessor derives from the fact that the transistors and interconnecting wires used to implement the processor on an integrated circuit chip are visible only under a microscope. Originally, microprocessors were less capable than their larger brethren, but recently their capabilities and speed exceed those of many of the large mainframes of the 1960s.

Because the memory devices and switching circuits used to build these machines were capable of reliably storing and processing only two values, binary numbers were and still are used to encode the machine's instructions, memory addresses, and data to be processed, including integers, alphabetic and numeric characters, and approximations to real numbers using floating-point representations. The representation of each item thus consists of an n-tuple of binary digits called bits (with, in general, a different choice of n for each type of encoded quantity).

In the early 1960s, International Business Machines (IBM) Corporation's design of the 360 series (or family) of computers, all of which were (in an upward compatible way) to execute the same instructions on the same kinds of data using different hardware implementations that would achieve different price versus performance (speed) market niches, brought with it the concept of *instruction set architecture* or simply *computer architecture* [3], [8]. Computer architecture encompasses the specification of an instruction set and the hardware units that implement the instructions. These specifications include

the set of instructions themselves using mnemonics and symbols suitable for a programmer to assemble programs for execution on the hardware, the encoding of the instructions in binary form suitable both for the hardware to interpret and for someone or some program to translate between symbolic and binary forms, the size of the memory space in which programs and data can be stored, and the number of programmer visible registers and their lengths for holding at least temporarily the n-tuples of bits being processed. In other words, computer architecture specifies the programmer's view of the machine and defines the hardware/software interface. For instance, a machine lacking hardware to directly implement floating-point arithmetic requires that programmers implement floating-point arithmetic in the software.

With the advent of the microprocessor on a chip, an entire digital computer system could be built from one or a few chips that included all of the functionality to execute machine instructions, store and retrieve data in memory devices, acquire outside world inputs, output commands and results, and make all of the components communicate with each other and interface to the outside world. One then has the option of building a user-programmable/reprogrammable desktop or portable personal computer or to embed the computer in another device or machine controlled by the embedded computer. The designer of the device or machine in which the computer is embedded writes the program to control the device, and the user of the device interacts with this program by, e.g., setting dials or pushing buttons.

In this brief introduction to computer architecture, we will specify the instruction set for a small hypothetical computer, discuss the hardware components needed to specify the microarchitecture on which the machine instructions will be interpreted, specify the microprogram [4], [9] for interpreting the original machine instructions, and then specify some more highly encoded microinstructions that can be directly interpreted by the hardware. This will take us from the concept of a complex instruction set computer (CISC), whose instructions are interpreted by a lower set of programs written by a microprogrammer, to a reduced instruction set computer (RISC), whose machine instructions are themselves (for the most part) microinstructions.

2 Functional Units

The basic functional units in a digital computer are the central processing unit (CPU), a memory system, an input unit for obtaining data from the outside world, and an output unit for sending data, results of computations, and commands to the outside world. The CPU contains an arithmetic and logic unit (ALU) and a control unit. The ALU is used to perform arithmetic and logical operations on data. The control unit fetches program instructions from the memory and issues control signals to interpret and execute the instructions using the available hardware components. The CPU also contains at least one, but usually several, memory devices called registers that the pro-

grammer can manipulate either directly or implicitly to hold data while it is in the CPU. There are also some hardwired constants and hidden registers (called the scratch-pad) that can be used only by the machine designer or microprogrammer to accomplish the task of interpreting the programmer visible instruction set on the underlying hardware components.

When program instructions and operand data are stored in the same memory, as shown in Fig. 1, the structure is called a *von Neumann architecture* [3]. When program instructions are stored in a memory separate from the memory in which data operands are stored, as in Fig. 2, the structure is called a *Harvard architecture* [14].

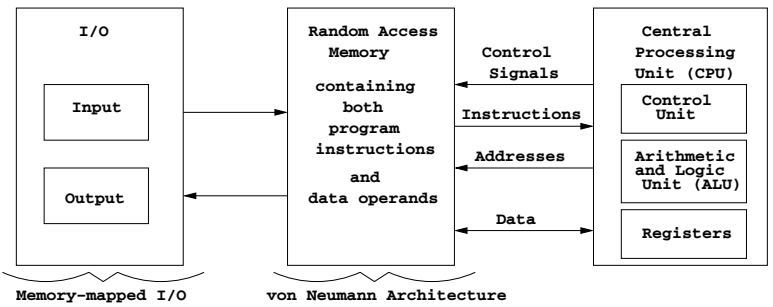


Fig. 1. Basic functional units organized in von Neumann architecture with memory-mapped I/O

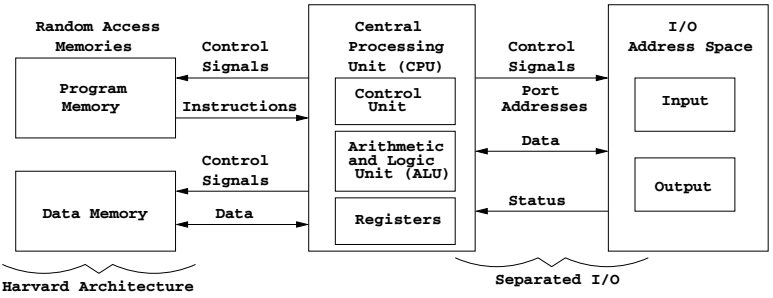


Fig. 2. Basic functional units organized as Harvard architecture with separated I/O

Two ways of handling input/output are displayed in these two figures, namely, memory-mapped I/O and separated I/O. These two schemes for handling input and output could be exchanged in the two figures without changing the designation of von Neumann or Harvard. In memory-mapped I/O, device registers are part of (and therefore consume some of) the memory address space. The I/O devices' data, status, and command registers are manipulated

by having the CPU fetch and execute load, store, and move instructions. In separated I/O, the devices' data, status, and command registers form their own address space separate from the memory address space, and the CPU must have input and output instructions that (when executed) cause the I/O device ports (instead of memory locations) to respond.

2.1 Registers and memory

A register is a device for recording and remembering some encoded quantity. The ten position wheels in a U.S. automobile's odometer record and remember a finite length string of decimal digits that display the number of miles the automobile has traveled, where translation to the number of miles driven is produced by the ratios of gears that count revolutions of the transmission output shaft. In most such automobiles, when the odometer registers 99,999.9 miles and the drive wheels turn an additional 0.1 mile, the odometer reads 00,000.0 miles. In other words, finite length registers enforce modular arithmetic where, in this case, multiples of 100,000.0 miles are equivalent to zero miles because there is no wheel to hold the 1 that carries out the left end of the register. The same is true for n-bit registers used in digital computers; one n-bit register can represent exactly 2^n quantities encoded with binary numbers in the range $\{0, ..., 2^n - 1\}$ independent of what those numbers represent.

Memory can be viewed as a collection of n-bit registers, each with a unique k-bit binary address, similar to a linear array of mailboxes in an apartment building or post office. Each mailbox is labeled with a number known as its absolute address and each, for example, can hold in length a number 10 envelope on which n bits can be written. The contents of mailbox number i is the n-bit number written on the envelope inside mailbox i. Postal clerks and apartment dwellers usually write an individual's name on some kind of tape and paste it on the mailbox door. This is an example of binding the symbolic address represented by the individual's name to the absolute address known as the mailbox number. In assembling a computer program for storage in the memory, each symbolic (variable) name or label is a symbolic address that must be bound to an absolute memory address. This address binding is usually handled by a translating program called an assembler or finally by another program called a linkage editor [12].

We shall designate CPU registers by a symbolic name (e.g., reg), and this name written in parentheses as "(reg)" represents the contents of register reg. We shall denote the contents of a memory location whose address corresponds to the symbolic name z as m[z], and the contents of the memory location whose address is in the register named reg as m[(reg)].

It is desirable to have memory constructed so that it takes the same amount of time to access any randomly selected address location. This is called a random access memory, as opposed to a sequentially accessed memory laid out on a reel of magnetic tape. Obviously, it takes longer to read down the tape to find data located near the hub end of the tape reel than it does to

find something at the front of the tape that is read first. Sequential access devices such as tapes, semi-random access devices such as magnetic disks and drums, and other similarly accessed devices are treated as I/O devices rather than memory. As long as stray magnetic fields are prevented (or diverted), magnetic I/O devices provide a means for remembering data when power is turned off and, thus, provide one way to achieve a non-volatile memory capability; optically sensed compact disks (CDROMs) are another means for doing so.

2.2 Hardware components

Registers are built from static memory devices known as latches and flip-flops [7], and these in turn are built using logic gates. The logic gates were originally built using vacuum tubes, but nowadays are built from transistors laid out on an integrated circuit chip.

Boolean algebra, gates, and latches

The logic gates represent logic 1 and logic 0, each with a range of voltages or currents. Table 1 specifies fundamental Boolean algebra [7] operations for both the logic gates, whose symbols are shown in Fig. 3, and bitwise logical operations on n-bit operands. For instance, given n-bit operands $\mathrm{A} = [\mathrm{A}_{n-1}, \ldots, \mathrm{A}_i, \ldots, \mathrm{A}_0]$ and $\mathrm{B} = [\mathrm{B}_{n-1}, \ldots, \mathrm{B}_i, \ldots, \mathrm{B}_0]$, $\mathrm{A} \wedge \mathrm{B} = [\mathrm{A}_{n-1} \wedge \mathrm{B}_{n-1}, \ldots, \mathrm{A}_i \wedge \mathrm{B}_i, \ldots, \mathrm{A}_0 \wedge \mathrm{B}_0]$, and similarly for the other bitwise logical operations found in computer instruction sets.

Table 1. Boolean logic functions

Inputs		Output functions					
		NOT	AND	OR	NAND	NOR	XOR
A	B	$\overline{\mathrm{A}}$	A∧B	A∨B	$\overline{\mathrm{A} \wedge \mathrm{B}}$	$\overline{\mathrm{A} \vee \mathrm{B}}$	A⊕B
0	0	1	0	0	1	1	0
0	1	1	0	1	1	0	1
1	0	0	0	1	1	0	1
1	1	0	1	1	0	0	0

A basic gated D-latch is formed by the feedback coupling of two NAND gates, which themselves form a set/reset (S/R) latch. If the set input is asserted, then after some propagation delay a logic 1 is stored at the Q output and its complement, logic 0, at the $\mathrm{Q}' = \overline{\mathrm{Q}}$ output. (Values at Q and Q′ are reversed if the reset input is asserted.) The latch retains these values so long as power is applied to the transistors implementing these gates; however, if power is lost, data stored in the latch evaporates and is forgotten. If the set and reset inputs are the complements of each other as provided by the input

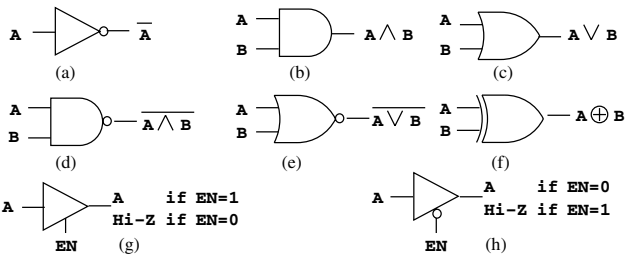


Fig. 3. Symbols for Boolean logic gates: (a) NOT, (b) AND, (c) OR, (d) NAND, (e) NOR, (f) XOR, and three-state non-inverting buffer amplifiers (g) and (h) with enable input (EN) and high-impedance (Hi-Z) output

inverter (NOT gate) connected to the D line, then the latch would act as a high gain combinational circuit following the D input. By placing two additional NAND gates controlled by the signal labeled "clk" as shown in Fig. 4, then the state of the feedback coupled NAND gates can change in response to the signal on the D input only when the clk line goes to logic 1 (we assume that D remains at its static 1 or 0 value until after the clk signal goes back to 0. While clk is logic 0, the latch portion remembers the last D value sampled while clk was equal to 1.

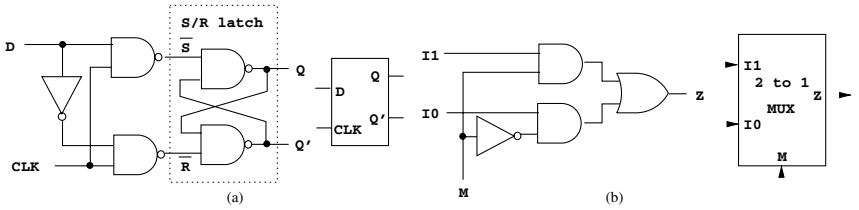


Fig. 4. (a) D-latch and symbol and (b) 2 to 1 multiplexer (MUX) and symbol

Another component shown in Fig. 4 is a 2 to 1 multiplexer (MUX) that acts as a two-way switch controlled by the selection input M. Input I_0 appears at the output if M = 0 and input I_1 appears at the output if M = 1. A 2 to 1 multiplexer that switches two n-bit sources to a single n-bit output can be built using n 2 to 1 multiplexers (one per output bit line), all controlled by the same selection signal M.

Registers and buses

Fig. 5 displays an 8-bit register built from D-latches and connected to input bus wires (the C-bus) and to two output buses (the A-bus and the B-bus). Connections to the A and B buses are controlled by three-state buffers that connect latch outputs to bus wires or leave connections open circuited (i.e., in

the Hi-Z state) under control of output enable signals OE-A and OE-B (see Fig. 3). The shorthand functional specification of a similar 16-bit register is shown in Fig. 6. The schematic for the 8-bit register in Fig. 5 would have the 8-bit A, B, and C buses labeled with an 8 to indicate that each line consists of 8 wires, instead of only 1 wire (unlabeled) or of 16 wires, as in Fig. 6.

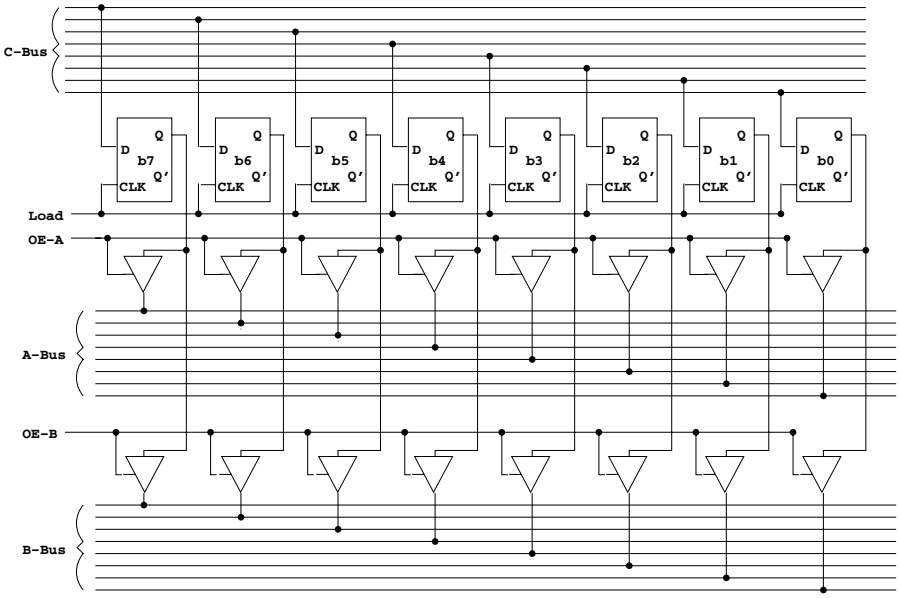


Fig. 5. Eight-bit register and bus connections

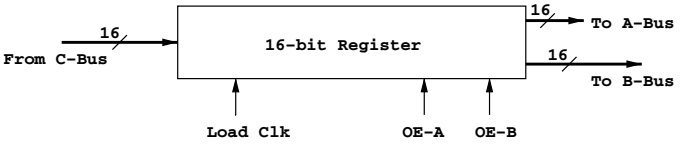


Fig. 6. Sixteen-bit register schematic

Arithmetic and logic unit

Fig. 7 presents the combinational logic circuitry and a shorthand schematic for a simple ALU that processes one bit of an n-bit number (or of two n-bit numbers) depending on which function is selected by the decoder section in response to the code on inputs F1 and F0. Connecting 16 such modules

together by feeding the carry out signal of one module to the carry in signal of the next module (all controlled by the same code on F1 and F0) provides a 16-bit ALU that can add two 16-bit numbers ($F_1F_0 = 10$), compute the bitwise logical AND of two n-bit numbers ($F_1F_0 = 01$), produce the bitwise logical complement (which corresponds to the arithmetic 1's complement [7]) of the n-bit number on the A inputs ($F_1F_0 = 11$), or pass the bits on the A inputs through unchanged ($F_1F_0 = 00$).

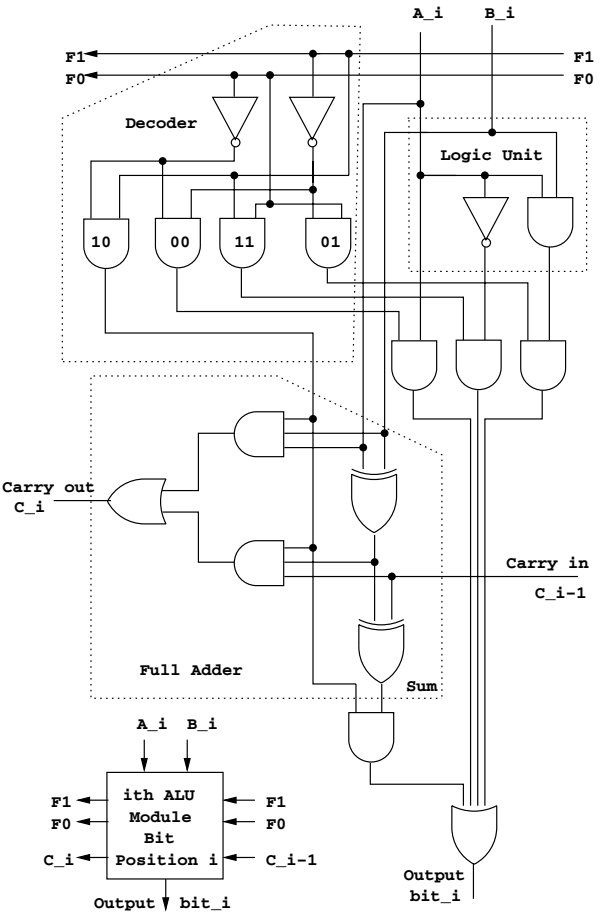


Fig. 7. Example ALU, i^{th} module

Shift unit

Fig. 8 shows how a simple 1-bit logical left or right shift unit can be built from logic gates. Only three of the possible four outputs from a 2-bit decoder

that responds to shift code inputs S1 and S0 are implemented. The three possibilities are left shift ($S_1S_0 = 10$), right shift ($S_1S_0 = 01$), and no shift ($S_1S_0 = 00$), which passes the data straight through unshifted. This 4-bit shift unit can easily be extended to shift 16-bit operands by inserting between D2 and D3 12 more repetitions of the structure for D2 (or D1).

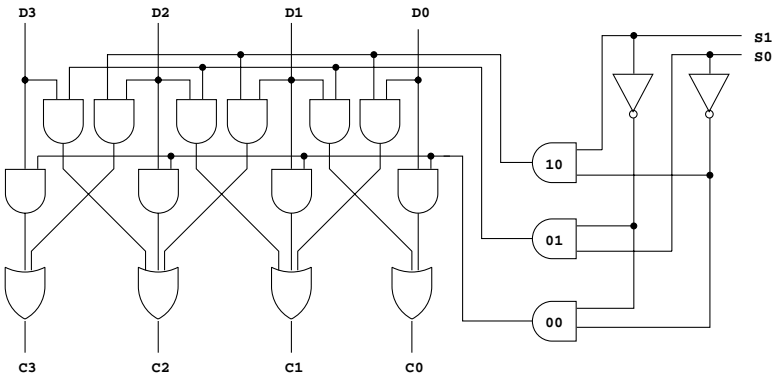


Fig. 8. Combinational logic shifter, 1-bit left or right, or no shift

System clocking unit

Because data will be moved from one register to another or through the ALU and shifter en route to the destination, a means of controlling when the data are latched (to avoid race conditions) must be provided to sequence the signals emanating from the control unit. This mechanism is usually provided by a multi-phase clock circuit. A four-phase non-overlapping clock signal is shown in Fig. 9.

Memory interface units

The control unit needs some buffer registers to hold bits while waiting for a memory access to occur. It needs a memory address register (MAR) to hold the address of the memory location being accessed so that the memory controller can continuously view the address it is in the process of decoding. It also needs a memory buffer register (MBR) into which to place the data to be written to a memory location or to hold data being read out of a memory location until the control unit is ready to copy the data to another register or do something else with those bits. Discrete signals for loading information into the MAR and MBR are needed, as are read (rd) and write (wr) control signals to activate the appropriate response by the memory controller and to turn around the 16-bit bidirectional data bus to memory. See the example MBR control structure in Fig. 10.

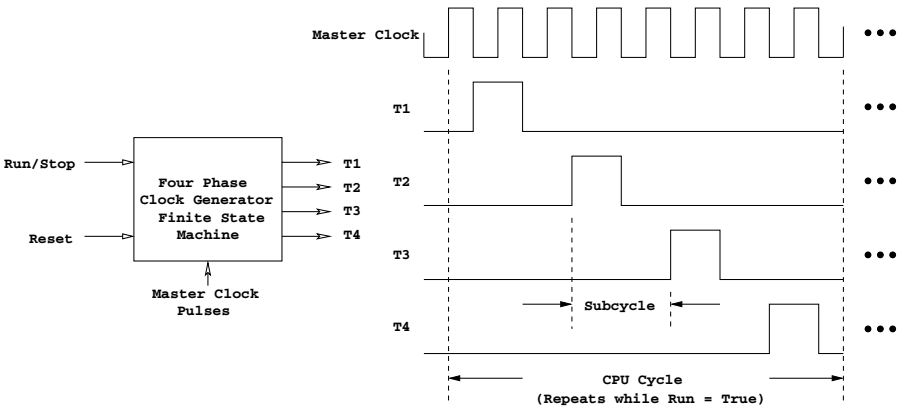


Fig. 9. Four-phase clock

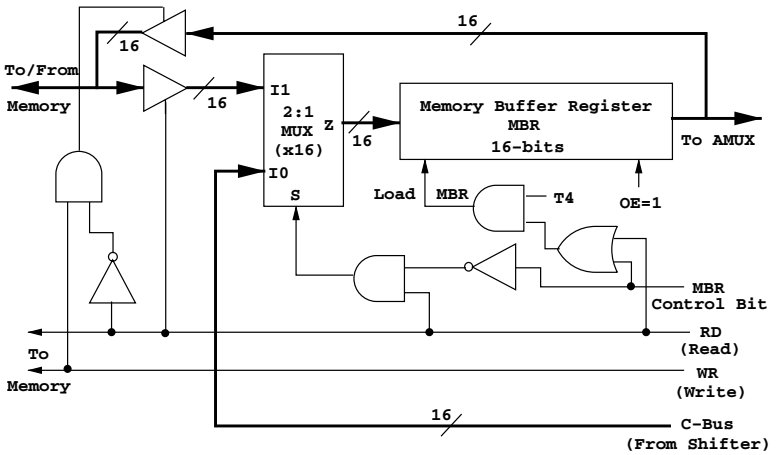


Fig. 10. Memory buffer register

3 Example Architecture

In specifying an example 16-bit word length computer architecture we must first specify the maximum size of the memory for which programmers can write programs. Let us assume that size is $2^{16} = 65,536$ memory locations, each containing a 16-bit word so that each location is designated by a 16-bit address. Word length typically corresponds to main memory data bus width and to the width of registers in the CPU. One could view the left and right halves of each 16-bit word as two 8-bit bytes and then make the memory byte addressable, but for simplicity we shall not do so here. Our example architecture is word addressable.

The next item to be specified is the number and types of user-visible CPU registers. Instructions in the instruction repertoire must be provided that allow

the programmer to manipulate the contents of these registers. We will assume the user-visible register set to be an accumulator register (AC), an index register (XR), a multiplier/quotient (MQ) register, a general purpose register (GR), and two memory pointer registers, a stack pointer (SP) register that points to the current top of the stack, and a base pointer (BP) register that points to a fixed location in each stack frame [2] in case the SP register contents change while a particular stack frame is being examined. The program counter (PC) register is used to point to the next instruction to be fetched into the CPU when execution of the current instruction is complete. The programmer is provided instructions to save and restore the contents of the PC register and to change the flow of control by modifying its contents. These registers and their connections to each other and other CPU functional units on 16-bit buses are shown in the datapath layout for the example microarchitecture in Fig. 11.

Fig. 11 also shows hardwired constants and scratch-pad registers needed by the control unit to accomplish its mission of fetching machine instructions from memory, decoding, and executing them. The instruction register (IR) holds the instruction being decoded and interpreted for execution. The 16-bit hardwired constants in registers 0 and +1 are obvious, all zeros in one case and 15 zeros on the left and a rightmost 1 in the second; but the contents of the −1 register are minus 1 represented in 2's complement, which is 16 ones. The XMASK register contains 4 zeros on the left followed by 12 ones on the right, and the YMASK register contains 8 zeros on the left followed by 8 ones on the right. In addition to the MAR and MBR buffer registers there are three scratch-pad registers (S1, S2, and S3) that can be used by the control unit. These constants and scratch-pad registers are not visible to the machine or assembly language programmer, but they are visible to the machine designer, who himself may use a form of programming called microprogramming [4] to implement the instruction set architecture. The A-latch, B-latch, and C-latch registers are used to hold output values constant while they propagate through combinational logic circuits and appear for latching at the input to one of the other registers. The time at which various registers latch their values is also indicated. The N signal emanating from the ALU and going to the micro-sequencing logic unit is a copy of the high order bit (bit 15) coming out of the ALU, which corresponds to the sign bit position. The Z signal is the output of a NOR gate that takes the complement of the result of ORing together all 16 ALU output bits, and is thus used to detect when the ALU outputs are all zeros.

Instruction set

In designing an instruction set the most basic computation one wishes to perform is to add the contents of two memory locations and to store the result in a third location, which can be expressed using variable names as C = A + B. The format for such an instruction would require five fields, one

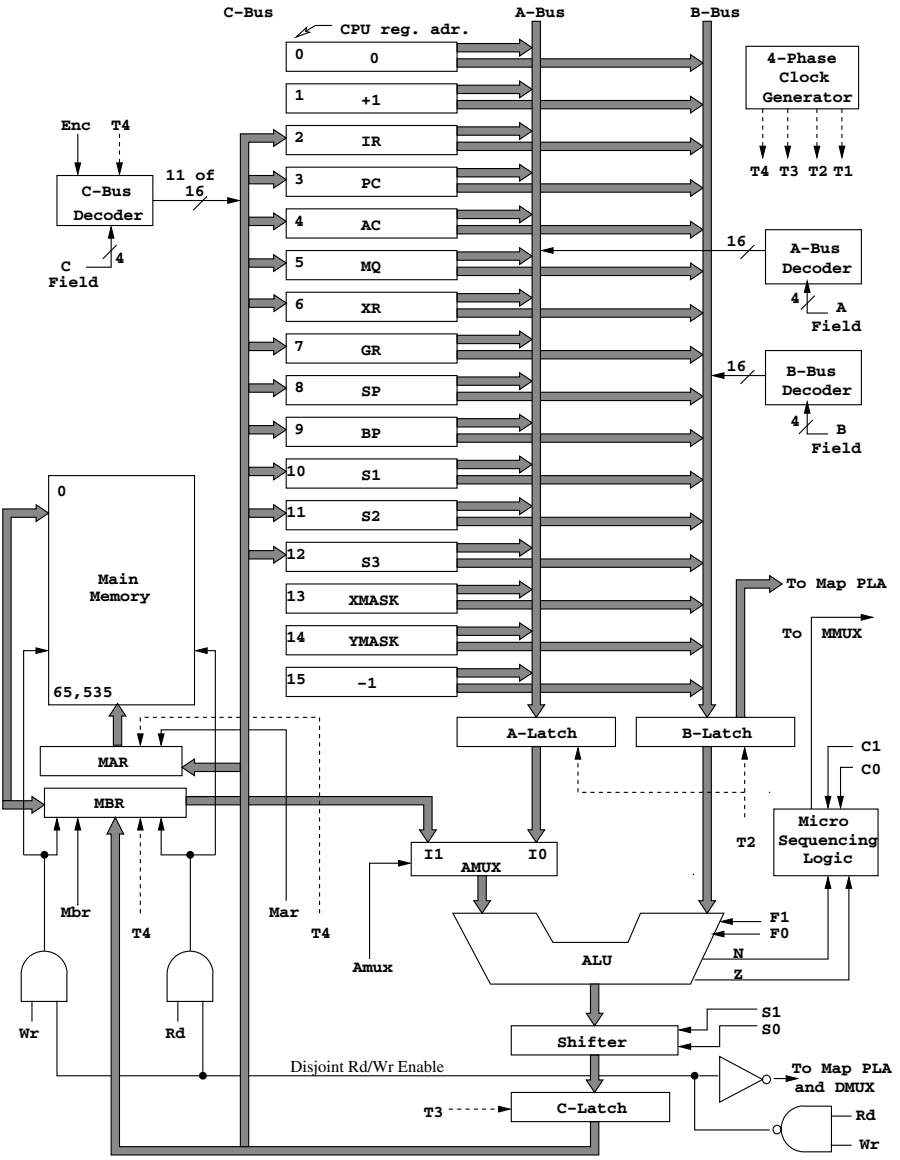


Fig. 11. Example microarchitecture datapath

to encode all of the desired operations including addition, three fields at the rate of one each for the memory addresses A, B, and C, and one more for the address of the next instruction to be fetched. This is known as a four address format. The problem with this format is that, in our case, each instruction would require $64 = 4 \times 16$ address bits. As most programmers write sequences of machine instructions that are to be fetched and executed one after the other until some decision is made to go to another part of the program, the easiest way to save 16 bits would be to remove the next address field, provide the machine with an automatically incrementing program counter register, and give the programmer jump and branch instructions that can manipulate the value in the program counter register using addresses from one or more of the other three address fields. This results in a three address format, but doing so still requires, in our case, 48 bits of address in each instruction. Making one of the operands implicitly either one of the sources or the destination for the result of the operation produces a two operand format in which the result of the operation overwrites one of the operands. Up to this point all of the instructions operate memory to memory (except for those that manipulate the program counter register).

Giving the programmer a CPU register such as an accumulator register, and instructions to load the accumulator from memory and store the contents of the accumulator back into memory produces a one address format wherein the accumulator is both an implicit source and destination for two operand instructions such as addition. If there is more than one CPU register for operands, then a register designator field, shorter than a memory address field, in the instruction is needed to encode which register is either the source or destination or both. This is sometimes called a one and one-half address format. Both forms usually comprise memory-to-register (and vice versa) formats. Register-to-register operations such as adding the contents of two registers and leaving the result in one of them could be handled by providing operation codes that permit the memory address field to also contain a much shorter register designator address. Some of these instructions are illustrated in Table 2.

Table 2. Sample instruction formats; operand C is destination

Three address	Two address	One address
ADD C, B, A	MOVE C, A	LOAD A
	ADD C, B	ADD B
		STORE C

A minimal instruction set would include data movement instructions to load and store CPU registers and to move data between programmer-visible registers, a subtract instruction, and a conditional transfer of control instruction. Subtract can be used to generate the bitwise logical complement of a bit

pattern, to effectively multiply a number by minus one by subtracting that number from zero, and to perform addition. Left logical shift, which corresponds to multiplying a number by two, can be obtained by adding a number to itself. Left circular shift can be obtained from addition by first checking if the leftmost bit is one or zero using the conditional transfer of control instruction, then adding the number to itself, and then adding to the logically left shifted value the constant one (or not) depending on the result of the conditional transfer of control test. Right circular shift by k bits can be obtained for n-bit values by a left circular shift of $n - k$ bit positions. Instructions in addition to the minimal instruction set provide convenience to the programmer or capabilities that would otherwise be tedious to program with fewer instructions.

The instructions chosen for our example instruction set architecture ISA-1 that will operate on the datapath in our example CPU and memory organization are shown in Tables 3 and 4, and their binary encodings are shown in Table 5. A register transfer language (or notation) is used to specify their actions in Tables 3 and 4. The colon-equal sign is similar to the notation for an assignment statement in Algol or Pascal and acts as a back arrow that points to the destination register to be written with the results of the action. If more than one register transfer statement is listed separated by semicolons, then the actions occur sequentially in left to right order.

The binary operation codes assigned to the ISA-1 instructions are specified in Table 5, which also lists the operand fields whose binary values (x, y z, or d) are to be filled in by the programmer. Assembler programs are usually written to permit the assembly language programmer to specify the operation using the symbolic mnemonic shown and to specify the content of an instruction's operand field either symbolically, or as a decimal, hexadecimal, or octal constant. The assembler program then fills in the binary operation code and calculates the appropriate binary value (x, y, z, or d) to fill into the operand field when it translates the source program into binary machine language.

Addressing modes

Examining the instructions in Table 5 that refer to memory locations, we see that their operation codes consume 4 of the 16 bits in the instruction word, leaving only 12 remaining bits in which to specify a 16-bit memory address. That is why these instructions' assembly language mnemonics end in X: to specify that the content of the 16-bit index register XR will be added to the signed 2's complement 11-bit offset in the instruction itself to form the final 16-bit memory address to be accessed, known as the *effective address*. This mechanism is known both as indexed addressing and displacement addressing [2], [3]. This mode also provides a register indirect mode by setting the offset field in the instruction to zero. Because reference to any arbitrary memory location requires that register XR first be loaded with an appropriate value,

Table 3. Example instruction set architecture (ISA-1) instruction repertoire

ISA-1 instruction repertoire part 1 of 2		
Assembly language	**Instruction**	**Meaning or action**
`LXRI x`	Load xr immediate	xr:=x (0≤x≤4095)
`LACX x`	Load ac reg	ac:=m[x+(xr)] (-2048 ≤x≤2047)
`STAX x`	Store ac reg	m[x+(xr)]:=(ac) (-2048 ≤x≤2047)
`ADDX x`	Add to ac	ac:=(ac)+m[x+(xr)] (-2048 ≤x≤2047)
`SUBX x`	Subtract	ac:=(ac)−m[x+(xr)] (-2048 ≤x≤2047)
`LEAX x`	Load effective address	ac:=x+(xr) (-2048 ≤x≤2047)
`JMPX x`	Jump	pc:=x+(xr) (-2048 ≤x≤2047)
`CALX x`	Call procedure	sp:=(sp)−1;m[(sp)]:=(pc); pc:=x+(xr) (-2048 ≤x≤2047)
`BREZ x`	Branch if zero	**if** (ac)=0 **then** pc:=(pc)+x (-2048 ≤x≤2047)
`BRLZ x`	Branch if negative	**if** (ac)<0 **then** pc:=(pc)+x (-2048 ≤x≤2047)
`BALX x`	Branch & link register	xr:=(pc); pc:=(pc)+x (-2048 ≤x≤2047)
`LODL x`	Load local	ac:=m[x+(bp)] (-2048 ≤x≤2047)
`STOL x`	Store local	m[x+(bp)]:=(ac) (-2048 ≤x≤2047)
`ADDL x`	Add local	ac:=(ac)+m[x+(bp)] (-2048 ≤x≤2047)
`SUBL x`	Subtract local	ac:=(ac)−m[x+(bp)] (-2048 ≤x≤2047)
`LABI y`	Load byte immediate	ac_{15-8}:=0, ac_{7-0}:= y
`INXR y`	Increment xr reg	xr:=(xr)+y (0≤y≤255)
`INSP y`	Increment sp reg	sp:=(sp)+y (0≤y≤255)
`DESP y`	Decrement sp reg	sp:=(sp)−y (0≤y≤255)
`LXRU z`	Load xr upper	xr_{15-12}:=y (0≤z≤15)
`RACR z`	Rotate ac z-bits right	ac_i:=($ac_{i \oplus z \bmod 16}$), (0≤z≤15)
`DROR z`	Rotate double register ac:mq z-bits right	ac:mq_i:=(ac:mq_{i+z}), (0≤i≤15-z), ac_i:=(mq_{i-16-z}), mq_i:=(ac_{i-16-z}), (16-z≤ 1i≤15), (0≤z≤15)
(continued)		

two immediate mode addressing instructions are provided for loading a constant value into this register immediately, LXRI to load the low order 12 bits and zero the high order (leftmost) 4 bits and LXRU to load the high order 4 bits without modifying the low order 12 bits. Two other instructions can also be used to obtain 16-bit values. The LAWI d, where d represents a 16-bit value in the word immediately following the word containing the LAWI opcode, loads the AC register. This must be followed by a MVAX instruction to copy the value into the XR register. The BALX instruction copies the PC register contents into the XR register and branches (jumps) relative to the PC contents plus the signed 11-bit offset in the low order 12 bits of the BALX instruction. Executing BALX 0 causes the instruction in the next sequential

Table 4. ISA-1 instruction repertoire (continued)

ISA-1 instruction repertoire part 2 of 2		
Assembly language	**Instruction**	**Meaning or action**
`LAWI d`	Load ac immediate with data word d	ac:=d (0≤d≤65535)
`RETN`	Return	pc:=m[(sp)];sp:=(sp)+1
`PUSH`	Push ac onto stack	sp:=(sp)−1;m[(sp)]:=(ac)
`POP`	Pop ac from stack	ac:=m[(sp)];sp:=(sp)+1
`PSHB`	Push bp onto stack	sp:=(sp)−1;m[(sp)]:=bp
`POPB`	Pop bp from stack	bp:=m[(sp)];sp:=(sp)+1
`MVSB`	Copy sp to bp	bp:=(sp)
`MVBS`	Copy bp to sp	sp:=(bp)
`SWAS`	Swap ac & sp	tmp:=(ac);ac:=(sp);sp:=(tmp)
`SWAX`	Swap ac & xr	tmp:=(ac);ac:=(xr);xr:=(tmp)
`MVAG`	Copy ac to gr	gr:=(ac)
`MVGA`	Copy gr to ac	ac:=(gr)
`MVAM`	Copy ac to mq	mq:=(ac)
`MVMA`	Copy mq to ac	ac:=(mq)
`INVA`	Complement ac	ac:=($\overline{\text{ac}}$)
`ANDG`	AND gr with ac	ac:=(ac)∧(gr)
`ADDG`	ADD gr to ac	ac:=(ac)+(gr)
`HALT`	Halt machine	stops fetching instructions

address to be fetched with the XR register pointing to it. BALX could also be used as a procedure call instruction to a target address relative to the current program counter contents while saving the return address in the XR register.

To support multiple and recursive calls, a procedure (subroutine or function) call instruction is provided that saves the return address in a memory location on the top of a stack pointed at by the SP register. If more values or addresses are passed to the called procedure than there are CPU registers available to hold them, then PUSH and POP instructions are provided to place these parameters on the stack prior to making the call. A stack frame or base pointer register, BP, is provided so that parameters in the local stack frame can be obtained or replaced using indexed addressing relative to the BP register instead of relative to the XR register. The LEAX instruction provides a means for loading an effective address into the AC register and passing it as a pointer parameter to a called procedure either in the AC register itself or by pushing its contents onto the stack.

Table 5. ISA-1 operation code assignments

ISA-1 instruction repertoire			
Opcode binary	**Assembly language**	**Opcode binary**	**Assembly language**
0000xxxxxxxxxxx	LXRI x	111110110000zzzz	DROR z
0001xxxxxxxxxxx	LACX x	1111110000000000	LAWI d
0010xxxxxxxxxxx	STAX x	dddddddddddddddd	
0011xxxxxxxxxxx	ADDX x	1111110000100000	RETN
0100xxxxxxxxxxx	SUBX x	1111110001000000	PUSH
0101xxxxxxxxxxx	LEAX x	1111110001100000	POP
0110xxxxxxxxxxx	JMPX x	1111110010000000	PSHB
0111xxxxxxxxxxx	CALX x	1111110010100000	POPB
1000xxxxxxxxxxx	BREZ x	1111110011000000	MVSB
1001xxxxxxxxxxx	BRLZ x	1111110011100000	MVBS
1010xxxxxxxxxxx	BALX x	1111110100000000	SWAS
1011xxxxxxxxxxx	LODL x	1111110100100000	SWAX
1100xxxxxxxxxxx	STOL x	1111110101000000	MVAG
1101xxxxxxxxxxx	ADDL x	1111110101100000	MVGA
1110xxxxxxxxxxx	SUBL x	1111110110000000	MVAM
11110000yyyyyyyy	LABI y	1111110110100000	MVMA
11110001yyyyyyyy	INXR y	1111110111000000	INVA
11110010yyyyyyyy	INSP y	1111110111100000	ANDG
11110011yyyyyyyy	DESP y	1111111000000000	ADDG
111110000000zzzz	LXRU z	1111111100000000	HALT
111110100000zzzz	RACR z		

dddddddddddddd is a 16-bit constant; in column 4 it is called d.
xxxxxxxxxxxx is a 12-bit constant; in column 2 it is called x.
yyyyyyyy is an 8-bit constant; in column 2 it is called y.
zzzz is a 4-bit constant; in columns 2 and 4 it is called z.

Control unit architecture with microprogramming

Fig. 12 provides a control structure for the microarchitecture's datapath in which the designer also implements datapath control signals using a form of programming, called microprogramming, in which microinstructions are stored in a control memory, called the control store to distinguish it for the time being from main memory. The control store has 256 words, and each word is 32 bits wide. The microinstructions, expressed mnemonically using a register transfer language, are then translated into 32-bit binary words whose 1's and 0's directly (or via decoders in functional units) control the data path for one CPU cycle comprising the four clock ticks provided by the 4-phase clock. Each rectangle in the diagram showing a dotted line labeled with one of the clock phases is or contains a register of D-latches that are loaded by that clock phase signal. The control store has an address register, called the microprogram counter (MPC), whose contents are decoded by the control store to select the next microinstruction to be latched into its buffer register,

called the microprogram instruction register (MIR), from which control signals emanate throughout the microarchitecture.

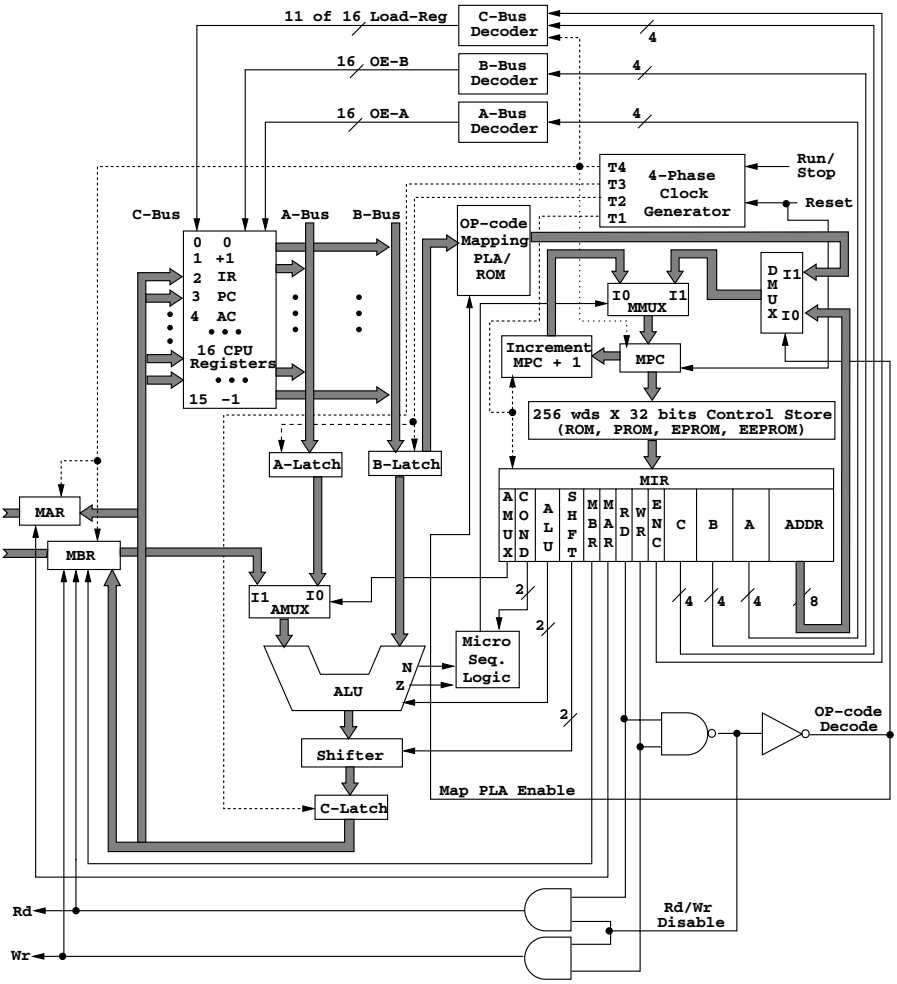


Fig. 12. Example microarchitecture control structure

We assume that the main memory can respond to read or write requests during the 4 clock ticks of one microinstruction cycle; otherwise, more reads or writes must be inserted consecutively in the microinstruction control stream if the memory is somewhat slower in responding. An alternate mechanism (not illustrated here) would be to provide a control signal sent by the memory controller to the CPU that indicates when the memory can provide the data requested or has completed writing the data sent by the CPU. This "ready" signal could then be sampled (if additional microoperations were available to

do so) in each subsequent microinstruction after a read or write request is either initiated or maintained to determine if the requested word is available or writing is complete.

The format of each microinstruction is shown in Fig. 13. The A and B fields are decoded to select either user-visible or other scratch-pad register contents to be gated onto the A and B buses, respectively. The C field selects one of the writable CPU registers as the destination into which C-bus contents are copied if the ENC bit is 1. If ENC is 0, then no CPU register in the scratch-pad copies the contents of the C-bus at T4, other than possibly the MAR or the MBR. Writing into the MAR and MBR from the C-bus is controlled by similarly named 1-bit control signals. Control bits RD and WR control reading and writing between main memory and the MBR. The AMUX bit controls the 2-way multiplexer by selecting either the contents of the A-latch (if 0) or the contents of the MBR (if 1) as the input data to the left input (A-bus side) of the ALU. ALU and shifter control signals are also shown and conform to decoder input selections shown in Figs. 7 and 8. Conditional and unconditional branching or jumps within the microprogram can be specified in parallel with ALU, shifter, and other datapath operations by the combination of the COND and ADDR fields. The microsequencing logic selects the source passing through the 2-way MMUX multiplexer by evaluating the Boolean expression: $\text{Mmux} = \overline{C_1}C_0N \vee C_1\overline{C_0}Z \vee C_1C_0 = C_0N \vee C_1Z \vee C_1C_0$. If the Mmux selection signal is 0, the incremented MPC value is taken; if Mmux = 1, the next ADDR field bits are taken, provided the DMUX selection input is 0.

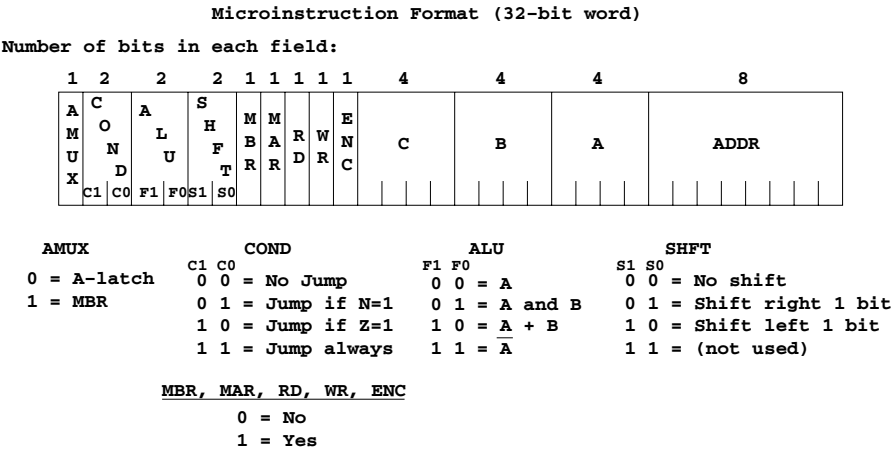


Fig. 13. Microinstruction control signals specification

The symbolic microoperation "decode" (not shown in Fig. 13) is translated into binary as $C_1C_0 = 11$ for an unconditional jump and both RD and WR equal 1 (which is meaningless to the main memory); this then sets (in Fig. 12)

the Mmux control input to 1 and the 2-way DMUX multiplexer selection input to 1 so as to select the output from the mapping programmable logic array (PLA) [2] as the input to the MPC. The mapping PLA decodes input from the B-latch, which should contain the operation code portion of the ISA-1 instruction in the IR; it forms a Boolean product term in the AND array that then generates in the OR array the 8-bit control store address where the first microinstruction of the execution sequence for that ISA-1 instruction is located.

A portion of the microprogram (comprising the first 43 microinstructions needed) to fetch, decode, and execute the ISA-1 instructions in Tables 3, 4, and 5 is shown in Table 6. [1] The execution cycle for each decoded ISA-1 instruction begins at the control store address whose line is labeled with a comment showing the assembly language mnemonic for the corresponding instruction (capitalized for emphasis). The corresponding operation code shown in the comment for each ISA-1 instruction lists the binary inputs to the mapping PLA, and the corresponding control store address labeled "Adr:" (shown in decimal at the left) is the corresponding output of the mapping PLA.

The instruction fetch cycle begins at control store address zero, and the "decode" and other microoperations shown at control store address 5 could be placed at control store address 3. Instead, we have chosen to create in scratch-pad registers s2 and s3 a couple of mask values needed during execution of memory reference instructions. This increases the length of the fetch cycle by 8 clock ticks for all instructions. In contrast, by generating these mask values in each instruction execution cycle that needs one or both of them, we would increase execution times for these instructions and require 23 more microinstructions, but we would shorten the fetch cycle for all instructions. The trade-off chosen here was to save 23 microinstructions by increasing the length of the instruction fetch cycle in the vain hope that the entire table could be displayed here.

If one were to write a microprogram assembler ([12]) in a high level language such as C for this register transfer language, then it would make life easier for the microprogrammer in modifying, correcting, or expanding the overall microprogram to be able to use symbolic addresses for ISA-1 instruction starting locations and other jump target labels. The resulting assembler symbol table (assuming the assembler produces absolute control store addresses) could then be used to program the mapping PLA.

Table 7 gives a few examples of register transfer language (RTL) statements in Table 6 (and an additional microinstruction from control store address 82 not shown in Table 6) translated into their corresponding 32-bit control store words whose 1-, 2-, 4-, and 8-bit fields are shown in decimal shorthand for each field. One need only convert the decimal value in each

[1] The complete microprogram requiring 127 control store words in addresses 0 through 126 can be found at World Wide Web URL: www.eng.umd.edu/∼silio/ISA1.

Table 6. Microprogram mpp-1 to fetch, decode, and execute ISA-1 instructions, part 1 of 3

Adr: Microinstruction	Comment
0: mar:=(pc);	fetch instruction
1: pc:=(pc) + (+1); rd;	update pc & read memory
2: ir:=(mbr);	load ir
3: s2:=rshift[(xmask) + (+1)];	(s2) = 0000100000000000
4: s3:=inv[(xmask)];	(s3) = 1111000000000000
5: s1:=band[(xmask),(ir)]; decode;	map control store address
6: xr:=(ir); goto 0;	0000 = **LXRI**
7: alu:=band[(s1),(s2)]; if z then goto 9;	0001= **LACX**
8: s1:=(s1)+(s3);	
9: mar:=(s1)+(xr);	
10: rd;	
11: ac:=(mbr); goto 0;	
12: alu:=band[(s1),(s2)]; if z then goto 14;	0010= **STAX**
13: s1:=(s1)+(s3);	
14: mar:=(s1)+(xr);	
15: mbr:=(ac);	
16: wr; goto 0;	
17: alu:=band[(s1),(s2)]; if z then goto 19;	0011= **ADDX**
18: s1:=(s1)+(s3);	
19: mar:=(s1)+(xr);	
20: rd;	
21: ac:=(ac)+(mbr); goto 0;	
22: alu:=band[(s1),(s2)]; if z then goto 24;	0100= **SUBX**
23: s1:=(s1)+(s3);	
24: mar:=(s1)+(xr);	
25: ac:=(ac)+(+1); rd;	
26: s1:=inv[(mbr)];	
27: ac:=(ac)+(s1); goto 0;	
28: alu:=band[(s1),(s2)]; if z then goto 30;	0101= **LEAX**
29: s1:=(s1)+(s3);	
30: ac:=(s1)+(xr); goto 0;	
31: alu:=band[(s1),(s2)]; if z then goto 33;	0110= **JMPX**
32: s1:=(s1)+(s3);	
33: pc:=(s1)+(xr); goto 0;	
34: sp:=(sp)+(-1);	0111= **CALX**
35: mar:=(sp);	
36: mbr:=(pc);	
37: wr; goto 31;	
38: alu:=(ac); if z then goto 40;	1000= **BREZ**
39: goto 0;	
40: alu:=band[(s1),(s2)]; if z then goto 42;	
41: s1:=(s1)+(s3);	
42: pc:=(s1)+(pc); goto 0;	

field to its binary value right justified and zero filled on the left for the corresponding number of bits in that field.

Table 7. Some RTL statements translated to 32-bit control store words

Register transfer language symbolic microinstruction statement	AMUX	COND	ALU	SHFT	MBR	MAR	RD	WR	ENC	C	B	A	ADDR
mar:=(pc);	0	0	0	0	0	1	0	0	0	0	0	3	00
pc:=(pc)+(+1);rd;	0	0	2	0	0	0	1	0	1	3	1	3	00
ir:=(mbr);	1	0	0	0	0	0	0	0	1	2	0	0	00
s2:=rshift[(xmask)+(+1)];	0	0	2	1	0	0	0	0	1	11	1	13	00
s3:=inv[(xmask)];	0	0	3	0	0	0	0	0	1	12	0	13	00
s1:=band[(xmask),(ir)];decode;	0	3	1	0	0	0	1	1	1	10	2	13	00
s1:=inv[(mbr)];	1	0	3	0	0	0	0	0	1	10	0	0	00
s1:=lshift[(s1)+(s1)];	0	0	2	2	0	0	0	0	1	10	10	10	00
alu:=(ac);if z then goto 40;	0	2	0	0	0	0	0	0	0	0	0	4	40
pc:=(s1)+(xr);goto 0;	0	3	2	0	0	0	0	0	1	3	6	10	00
sp:=(sp)+(+1);rd;goto 11;	0	3	2	0	0	0	1	0	1	8	1	8	11
s1:=(s1)+(-1);if n then goto 0;	0	1	2	0	0	0	0	0	1	10	15	10	00
ac:=rshift[inv[(ac)]];	0	0	3	1	0	0	0	0	1	4	0	4	00

Control with reduced microinstruction word width

Now consider an alternate control unit architecture that uses reduced word width microinstructions. The reduced word width is achieved by encoding the nine leftmost control fields in the 32-bit wide horizontal microinstruction format in Fig. 13 into one field and by eliminating the ADDR field, resulting in a microinstruction format (called mpp-2) that is only half as wide (16 bits). The consequence of encoding the microoperations in the microinstructions into 4-bit operation codes is that only one microoperation or register transfer at a time can be issued, thus reducing the parallelism provided by the microarchitecture's datapath. This occurs because the operation codes must be decoded into a 1-out-of-16 selection, as shown in Fig. 14. Two condition code latches for N and Z must also be supplied because now it is necessary to set the latches during one microinstruction (such as during an ADD, Boolean AND, or shift microinstruction) and then conditionally jump by testing them in a following microinstruction. Disposing of the ADDR field requires the introduction of a microprogram counter (already present) and both conditional and unconditional jump (micro)instructions whose target address can be formed by combining the R2 and R3 4-bit address fields into one 8-bit field.

Table 8 shows the reduced word width microinstruction encodings, their assembly language mnemonics, and their register transfer language specifi-

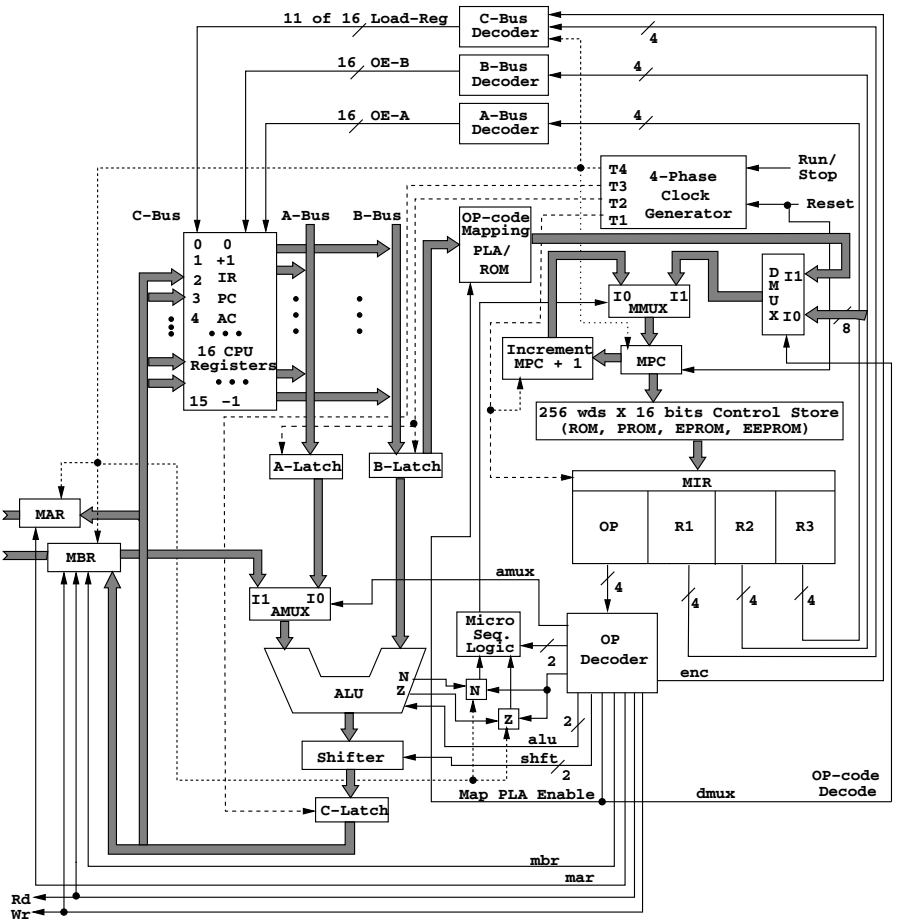


Fig. 14. Reduced word width microinstructions control structure

cations. Note that r is the 8-bit concatenation [r2r3] of the two 4-bit fields specified by r2 and r3 in the left-to-right order r2 followed by r3.

Table 9 specifies the control signals generated by the opcode decoder for each mpp-2 microinstruction opcode. A plus sign means the signal is asserted (i.e., set equal to 1); a blank means it is negated (i.e., set equal to 0). The "Latch NZ" control signal generated by the opcode decoder and the addition of the N and Z latches are the main differences between the mpp-2 and the mpp-1 microarchitectures; the remaining 12 control signals generated by mpp-2's opcode decoder are the same as those used to control the mpp-1 data path and, thus, perform the same functions as those specified in the microinstruction format mpp-1 (32-bit word). Each output column in Table 9 represents one control signal bit generated by an OR-gate whose input is the output of those decoder AND-gates showing a plus sign in that column.

Table 8. MPP-2 microinstructions, encodings, and meaning

MPP-2 opcodes			
Opcode binary	**Mnemonic & operands**	**Instruction**	**Meaning or action**
0000	MOVE r1,r3	Move register	r1:= (r3)
0001	AND r1,r2,r3	Boolean AND	r1:= (r2)∧(r3) = band[(r2),(r3)]
0010	ADD r1,r2,r3	Addition	r1:= (r2)+(r3)
0011	COMPL r1,r3	Complement	r1:= inv[(r3)]
0100	LSHIFT r1,r3	Left shift	r1:= lshift[(r3)]
0101	RSHIFT r1,r3	Right shift	r1:= rshift[(r3)]
0110	GETMBR r1	Store MBR in register	r1:= (mbr)
0111	TEST r3	Test register	**if** (r3)<0 **then** N:=1; **if** (r3)=0 **then** Z:=1
1000	LDMAR r3	Load MAR	mar:= (r3)
1001	LDMBR r3	Load MBR	mbr:= (r3)
1010	READ	Memory read	rd
1011	WRITE	Memory write	wr
1100	NJUMP r	Jump if N=1	**if** n **then go to** r
1101	ZJUMP r	Jump if Z=1	**if** z **then go to** r
1110	UJUMP r	Unconditional jump	**go to** r
1111	DECODE	Decode IR operation	**go to** map_PLA address

Table 9. MPP-2 decoding and control signals

MPP-2 control signals														
Opcode decimal	**Mnemonic**	**ALU F1**	**ALU F0**	**SHFT S1**	**SHFT S0**	**Latch NZ**	**AMUX**	**ENC**	**MAR**	**MBR**	**RD**	**WR**	**COND C1**	**COND C0**
0	MOVE					+		+						
1	AND		+			+		+						
2	ADD	+				+		+						
3	COMPL	+	+			+		+						
4	LSHIFT			+		+		+						
5	RSHIFT				+	+		+						
6	GETMBR					+	+	+						
7	TEST					+								
8	LDMAR								+					
9	LDMBR									+				
10	READ										+			
11	WRITE											+		
12	NJUMP													+
13	ZJUMP												+	
14	UJUMP												+	+
15	DECODE										+	+	+	+

On the road to RISC

In the 1960s IBM included both a read-only memory (ROM) control store with wide horizontal microinstructions and some writable control storage (static random access memory) for microdiagnostics in high end IBM 360 series machines. In low end machines, both the ISA instructions and the microinstructions were stored in the read/write magnetic core main memory. In later series machines, metal oxide semiconductor memory was used that held both ISA instructions and the control microprogram. The microprogram was loaded into memory from a magnetic floppy disk from the console by pressing a "load control store" button. Designers of some minicomputers (most of which had microprogrammed CPUs) began including some writable control store so that users could implement their own microprogrammed machine instruction, such as one instruction that might perform a fast Fourier transform. This led to the idea that the middle level microprogramming to interpret ISA machine instructions could be eliminated if only high level language compilers could compile all the way down to microinstructions. The first successful attempts to do this required microinstructions that could perform only one thing at a time, like the mpp-2 reduced word width microinstructions. So by designing an instruction fetch unit that would carry out the first three microinstruction words in Table 6 plus the instruction decode as a hardware component, the microprogram could be formatted in mpp-2 format and placed directly into main memory. In this way the PC becomes the MPC register used by the instruction fetch unit, the IR register takes the place of the MIR register, and the op decoder decodes the opcode field in the IR. The datapath, op decoder, and control signal gating are then all hardwired, and the mpp-2 microinstructions become the instruction set architecture with the entire register set visible to the programmer. This is basically what reduced instruction set computers (RISC) are: user-microprogrammed engines.

The problem with the reduced word width microinstructions in the mpp-2 format is that, because of the loss of parallelism, more of them are needed to accomplish the same tasks performed by horizontal mpp-1 microinstructions. For instance, the mpp-1 microinstruction "sp:=(sp)+(+1);rd;goto 11;" specified in execution of the POP instruction would need to be replaced by a sequence of three mpp-2 microinstructions; namely, "ADD sp,+1,sp; READ; UJUMP 11." Transforming to mpp-2 format the 127 words in the mpp-1 microprogram to interpret ISA-1 instructions (whose first 43 words are given in Table 6) would require 3 additional microinstructions in the fetch cycle and 71 additional microinstructions in the ISA-1 execution cycles. Thus, although in the mpp-2 format the microinstructions have become narrower, more of them are needed to do the same work, resulting in longer (taller?) microprograms; hence, the term "vertical microprogramming" for the mpp-2 format versus the term "horizontal microprogramming" for the mpp-1 format. At four clock ticks per additional microinstruction the same task would take longer on a ver-

tically microprogrammed machine than on a horizontally microprogrammed machine.

To recover some of the lost parallelism, a production line scheme, called a "pipeline," is used in all reduced instruction set architectures, but this too introduces problems. If four clock ticks are used to process one microinstruction, then a four stage production line could be used with one stage per clock tick. The instruction fetch unit is somehow operated to obtain a new microinstruction at each clock tick and feed it into the pipeline. Once the pipeline is filled, four separate microinstructions would be in various stages of execution simultaneously, thus regaining lost parallelism. The instruction fetch unit must predict from what location the next microinstruction will come (usually pointed at by the incremented program counter). Problems arise when conditional branch instructions that cannot be resolved until the last stage of production enter the pipeline. If the instruction fetch unit has guessed the branch target incorrectly, then the other three stages of production must be discarded and the correct instruction stream brought into the pipeline, causing delayed production of results. Much design effort has gone into ways of making better branch outcome guesses (i.e., predictions) to keep the ever-lengthening pipelined production lines busy producing good results. More recently, this work has been backed off onto compiler optimizers to reorganize the instruction sequence (without, hopefully, violating the programmer's intent) so as to schedule the instructions going into the pipeline in a way that will keep the production line busy ([3], [8], [11]).

If one increases mpp-2 format word length and main memory word length from 16 bits to 32 bits, then some more possibilities present themselves. The r1, r2, and r3 fields can be increased to 5 bits each and 32 addressable and user-visible registers can be included in the CPU. The width of the opcode field can also be increased from 4 bits to 6 or 8 bits. This allows more opcode possibilities and an instruction format that includes some bits in which to also specify immediate mode constants to provide addressing modes similar to those in the ISA-1 instruction set directly without the user writing sequences of mpp-2 like microinstructions. Main memory load and store instructions are also usually provided (along with opcodes for full and partial word transfers) so that users need not directly manipulate the MAR and MBR registers and issue read and write microoperations. These tasks are carried out and sequenced by the hardware decoder for load and store instructions. Instructions such as add, subtract, and other bitwise logical operations are restricted to manipulation of CPU registers. Instructions that specify floating-point operations may also be included in the instruction set, but these complex instructions are usually passed to an associated floating-point co-processor for execution.

Because main memory access time has not been able to keep up with improvements in CPU instruction cycle times, higher speed hidden buffer memory (actually static random access register memory called "cache memory") is usually placed between the CPU and main memory to provide on-average higher effective access time to memory words by keeping the more frequently

accessed words in the high speed buffer memory. If the desired word cannot be found in the cache memory, then some additional overhead time is incurred to obtain that word from main memory, place it in the cache, and hand it to the CPU.

The material in the references provides both additional information and more detailed discussions of concepts presented here. However, whichever instruction set is provided by designers, it is the task of the programmer to use those instructions to enable the computer to produce the desired results. More complex instructions may result in a sequence of fewer instructions to accomplish a given task. A simpler "reduced instruction set" format may require a longer sequence of instructions to do the same task, but hardware organization components such as pipelining and branch prediction to support on-average faster execution are also usually provided.

References

1. Carpinelli, J. (2001) *Computer Systems Organization & Architecture*. Addison-Wesley, Boston, San Francisco, New York.
2. Hammacher, C., Vranesic, Z., and Zaky, S. (2002) *Computer Organization*, 5th ed. McGraw-Hill, Boston, New York, San Francisco.
3. Hennessy, J. and Patterson, D. (2003) *Computer Architecture A Quantitative Approach*, 3rd ed. Morgan Kaufmann, Amsterdam, San Francisco, New York.
4. Husson, S. (1970) *Microprogramming, Principles and Practices*. Prentice-Hall, Englewood Cliffs, NJ.
5. Institute of Electrical and Electronics Engineers (1985) IEEE standard for binary floating-point arithmetic, ANSI/IEEE Std 754-1985. New York.
6. Kuo, S. and Gan, W-S. (2005) *Digital Signal Processors, Architectures, Implementations, and Applications*. Prentice-Hall Pearson, Upper Saddle River, NJ.
7. Mano, M. (1993) *Computer System Architecture*, 3rd ed. Prentice-Hall, Englewood Cliffs, NJ.
8. Patterson, D. and Hennessy, J. (1998) *Computer Organization and Design– The Hardware/Software Interface*, 2nd ed. Morgan Kaufmann, San Mateo, San Francisco.
9. Siewiorek, D., Bell, C.G., and Newell, A. (1982) *Computer Structures: Principles and Examples*. McGraw-Hill, New York, St. Louis, San Francisco.
10. Stallings, W. (2003) *Computer Organization and Architecture, Designing for Performance*, 6th ed. Prentice-Hall Pearson, Upper Saddle River, NJ.
11. Stone, H. (1993) *High-Performance Computer Architecture*, 3rd ed. Addison-Wesley, Reading, Menlo Park, New York.
12. Tanenbaum, A. (1999) *Structured Computer Organization*, 4th ed. Prentice-Hall, Upper Saddle River, NJ.
13. Wakerly, J. (1989) *Microcomputer Architecture and Programming, The 68000 Family*. John Wiley, New York, Chichester, Brisbane, Toronto.
14. Wolf, W. (2001) *Computers as Components, Principles of Embedded Computing System Design*. Morgan Kaufmann, San Francisco, San Diego, New York, Boston.

Real-Time Scheduling for Embedded Systems

Marco Caccamo,[1] Theodore Baker,[2] Alan Burns,[3] Giorgio Buttazzo[4], and Lui Sha[1]

[1] University of Illinois at Urbana Champaign, IL, U.S.A. {mcaccamo,lrs}@uiuc.edu
[2] Florida State University, FL, U.S.A. baker@cs.fsu.edu
[3] University of York, U.K. burns@cs.york.ac.uk
[4] University of Pavia, Italy buttazzo@unipv.it

1 Introduction

A real-time system is one with explicit deterministic or probabilistic timing requirements. Historically, real-time systems were scheduled by cyclic executives, constructed in a rather *ad hoc* manner. During the 1970s and 1980s, there was a growing realization that this static approach to scheduling produced systems that were inflexible and difficult to maintain. Building upon the seminal work of Liu and Layland [27], a successful effort was made to develop a practical theory of dynamic real-time scheduling, which led to the main body of fixed priority scheduling results reported here. In addition, there were notable and timely successes in the application of this theory to national high technology projects including a global positioning satellite software upgrade [17] and the International Space Station.

The successes in practice provided the momentum to revise the open systems standards and create a coherent set of hardware and software standards to support fixed-priority, theory-based real-time computing. Before these revisions, real-time computing standards were *ad hoc*. They also suffered from priority inversion problems, and had an inadequate number of priority levels to support real-time applications of fixed-priority scheduling. A standards-driven transformation of the real-time computing infrastructure started with solving the priority inversion problem in Ada [32], and in providing sufficient priority levels in Futurebus+. Today, all major open standards on real-time computing support fixed-priority scheduling.

Since the launch of a Real Time Systems Initiative by the United States Office of Naval Research in the 1980s, there has been an explosion of interest in real-time systems, and a growing amount of research and publications on the analysis of real-time scheduling. This chapter is based on the 25th year anniversary paper [31] for the IEEE Real Time Systems Symposium written by Sha et al. where the authors reviewed the key results in real-time scheduling

theory and the historical events that led to the establishment of the current real-time computing infrastructure. In the following sections, we briefly review the two most important areas of real-time scheduling theory: 1) fixed-priority scheduling, and 2) dynamic-priority scheduling. Finally, we examine some of the new challenges ahead.

2 Fixed-Priority Scheduling

The notion of priority is commonly used to order access to the processor and other shared resources such as communication channels. In real-time scheduling theory, priorities are principally applied to the scheduling of *jobs*. Each job has a *release time*, a *computation time*, and a *deadline*. The deadline can be expressed *relative* to the release time or as an *absolute* time. In priority scheduling, each job is assigned a priority via some policy. Contention for resources is resolved in favor of the job with the higher priority that is ready to run.

The phrase "fixed-priority scheduling" is generally applied to tasks. A *task*, sometimes also called a process or thread, is a potentially infinite sequence of jobs. A task is *periodic* if it is time triggered, with a regular release. The length of time between releases of successive jobs of task τ_i is a constant, T_i, which is called the *period* of the task. The deadline of each job is D_i time units after the release time. A task may also have an *offset*, from system start-up, for the first release of the task. A task is *aperiodic* if it is not periodic. An aperiodic task is *sporadic* if there is a bound on the load it may impose on the system. For a sporadic task there are constants C_i and T_i such that the sum of the compute times of all the jobs of τ_i released in any interval of length T_i is bounded by C_i. In one important case C_i is an upper bound on the compute time of each job and T_i is the minimum time between releases.

In fixed-priority task scheduling, all the jobs of a task have the same priority. Usually, the tasks are numbered so that τ_i has priority i, where the value one denotes the highest priority and larger integers denote lower priorities. Task-level priority assignments are known as "generalized rate-monotonic" scheduling because of historical reasons. The priority of each task is assumed to be fixed, but a system may still be analyzed under the assumption of fixed task priorities if the changes in priority only occur at major epochs, such as system "mode changes," or if the changes only apply to short time intervals, such as critical sections.

2.1 The Liu and Layland analysis

In 1973, Liu and Layland published a paper on the scheduling of periodic tasks that is generally regarded as the foundational and most influential work in fixed-priority real-time scheduling theory [27]. They started with the following assumptions:

(i) all tasks are periodic;
(ii) all tasks are released at the beginning of a period and have a deadline equal to their period;
(iii) all tasks are independent, i.e., have no resource or precedence relationships;
(iv) all tasks have a fixed computation time, or at least a fixed upper bound on their computation times, which is less than or equal to their period;
(v) no task may voluntarily suspend itself;
(vi) all tasks are fully preemptible;
(vii) all overheads are assumed to be 0;
(viii) there is just one processor.

Under this model, a *periodic* task is time triggered, with a regular release time. The length of time between releases of successive jobs of task τ_i is a constant, T_i, which is called the *period* of the task. Each job has a deadline D_i time units after the release time. A task is said to have hard deadline if every job must meet its deadline. A task may also have an *offset,* from system start-up, for the first release of the task.

Feasibility analysis is used to predict temporal behavior via tests which determine whether the temporal constraints of tasks will be met at run time. Such an analysis can be characterized by a number of factors including the constraints of the computational model (e.g., uniprocessor and task independence) and the coverage of the feasibility tests. Sufficient and necessary tests are ideal, but for many computational models such tests are intractable. Indeed, the complexity of such tests is non-deterministic polynomial (NP)-hard for non-trivial computational models. Sufficient but not necessary tests are generally less complex, but are pessimistic. Feasibility analysis is most successful in systems where the relative priorities of tasks (as in fixed-priority scheduling), or at least jobs (as with Earliest Deadline First scheduling), does not vary.

Liu and Layland's fundamental insight regarding the feasibility of fixed-priority task sets is known as the *critical instant theorem* [27]. A *critical instant* for a task is a release time for which the response time is maximized (or exceeds the deadline, in the case where the system is overloaded enough that response times grow without bound). The theorem says that, for a set of periodic tasks with fixed priorities, a critical instant for a task occurs when it is invoked simultaneously with all higher priority tasks. The interval from 0 to D_i is then one over which the demand of higher priority tasks $\tau_1 ... \tau_{i-1}$ is at a maximum, creating the hardest situation for τ_i to meet its deadline. This theorem has been proven to be robust. It remains true when many of the restrictive assumptions about periodic tasks listed above are relaxed.

Though further research would find more efficient techniques, the *critical instant theorem* provided an immediately obvious necessary and sufficient test for feasibility, i.e., to simulate the execution of the set of tasks, assuming they are all initially released together, up to the point that the lowest priority task

either completes execution or misses its first deadline. The task set is feasible if and only if all tasks have completed within their deadlines. The simulation only need consider points in time that correspond to task deadlines and release times. Since there are only $\lceil D_n/T_i \rceil$ such points for each task τ_i, the complexity of this simulation is $O(\sum_{i=1}^{n} D_n/T_i)$.

Based on the concept of critical instant, [27] proved a sufficient utilization-based condition for feasibility of when a set of tasks assigned priorities according to a *rate-monotonic* (RM) policy, that is, when the task with the shortest period is given the highest priority, the task with the next shortest period the second highest priority, etc. Liu and Layland proved that RM policy is the optimal static task priority assignment, in the sense that if a task set can be scheduled with any priority assignment, it is feasible with the RM assignment. Liu and Layland also showed that with this policy scheduling a set of n periodic tasks is feasible if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq n(2^{\frac{1}{n}} - 1). \tag{1}$$

For example, a pair of tasks is feasible if their combined utilization is no greater than 82.84%. As n approaches infinity, the value of $n(2^{1/n} - 1)$ approaches $\ln(2)$ (approximately 69.31%).

A common misconception is that with fixed-priority scheduling, and RM scheduling in particular, it is not possible to guarantee the feasibility for any periodic task set with a processor utilization greater than $\ln 2$. This bound *is* tight in the sense that there exist some infeasible task sets with utilization arbitrarily close to $n(2^{1/n} - 1)$, but it is *only a sufficient* condition. That is, many task sets with utilization higher than (1) are still schedulable. Lehoczky, Sha, and Ding [23] showed that the average real feasible utilization, for large randomly chosen tasks sets, is approximately 88%. The remaining cycles can be used by non-real-time tasks executing with background priorities. The desire for more precise fixed-priority schedulability tests, i.e., conditions that are necessary as well as sufficient, led to the feasibility analysis which is described later in this section.

It is also important to note that high utilization can be guaranteed by an appropriate choice of task periods. In particular, if task periods are *harmonic* (that is, each task period is an exact integer multiple of the next shorter period), then schedulability is guaranteed up to 100% utilization. This is often the case in practice in a number of application domains. Sha and Goodenough showed that by transforming periods to be nearly harmonic (with zero or small residues in the division of periods), the schedulability can be significantly improved [32]. For example, for two tasks with periods 10 and 15, the task with period 10 can be mapped into a new task with period 5 and half of the original execution time. The schedulability now becomes 100% because the residue in period division is now zero. This technique, called *period transformation*, can be done without changing the source code by means of one of the fixed-priority

aperiodic server scheduling techniques, such as the sporadic server. There is, however, some increase in system overhead when servers are used.

2.2 Further developments

The success of fixed-priority scheduling has come through the work of a large group of researchers, who extended the original analysis of [27] in a number of ways. The full story spans a very large number of publications. However, taking a historical point of view [3, 32], one can recognize within all this research a few principal threads:

(i) *exact feasibility analysis*—necessary and sufficient feasibility tests (based upon calculation of the worst-case response time of a task) permitted higher utilization levels to be guaranteed and led to further extensions of the theory, such as overrun sensitivity analysis and the analysis of tasks with start-time offsets;
(ii) *analysis of task interaction*—the introduction of the family of priority inheritance protocols enabled potential blocking to be bounded and analyzed;
(iii) *inclusion of aperiodic tasks*—the introduction of aperiodic servers permitted aperiodic tasks to be accommodated within the strictly periodic task model;
(iv) *overload management*—techniques for handling variations in task execution times made it possible to relax the requirement of known worst-case execution times and still ensure that a critical subset of the tasks will complete on time;
(v) *implementation simplifications*—the demonstration that only a small number of implementation priority levels is needed made it practical to apply fixed-priority scheduling more widely, including in hardware buses;
(vi) *multiprocessors and distributed systems*—analysis techniques were adapted to systems with multiple processors.

In the following sections we discuss these key results, together with the subsequent threads of research stemming from them.

2.3 Feasibility analysis

The utilization bound feasibility test described above is simple, both in concept and computational complexity. Consequently, it is widely recognized and is frequently cited. However, it has some limitations:

(i) the feasibility condition is sufficient but not necessary (i.e., pessimistic);
(ii) it imposes unrealistic constraints upon the timing characteristics of tasks (i.e., $D_i = T_i$);
(iii) task priorities have to be assigned according to the RM policy (if priorities are not assigned in this way, then the test is insufficient).

During the mid 1980s, more complex feasibility tests were developed to address the above limitations.

Lehoczky, Sha, and Ding [23] abstracted the idea behind the feasibility test (based on the *critical instant theorem*) for RM scheduling by observing that if a set of tasks is released together at time zero, the i highest priority task will complete its first execution within its deadline if there is a time $0 < t \leq T_i$ such that the demand on the processor, $W_i(t)$, of the i highest priority tasks is less than or equal to t; that is,

$$W_i(t) = \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j \leq t. \tag{2}$$

Since $\frac{t}{T_j}$ is strictly increasing except at the points where tasks are released, the only values of t that must be tested are the multiples of the task periods between zero and T_i. This test is also applicable to task sets with arbitrary fixed-priority orderings.

Concurrently, another group of researchers looked at the more general problem of determining the *worst-case response time* of a task, that is, the longest time between the arrival of a task and its subsequent completion. Once the worst-case response time of a task is known, the feasibility of the task can be checked by comparing its worst-case response time to its deadline.

A fixed-priority response-time analysis was developed in [21]. The algorithm of Joseph and Pandya computes the worst-case response time R_i of task τ_i as the least fixed-point solution of the following recursive equation:

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j. \tag{3}$$

Joseph observed that only a subset of the task release times in the interval between zero and T_i need to be examined for feasibility. That is, the preceding equation can be solved iteratively.

In 1982, the paper [24] considered fixed-priority scheduling of sets of tasks that may have deadlines that are less than their periods, i.e., $C_i \leq D_i \leq T_i$. Leung and Whitehead showed that the optimal policy for such systems, called *deadline monotonic* (DM) scheduling, is to assign tasks with shorter deadlines higher priorities than tasks with longer deadlines. RM and DM scheduling are the same when deadline equals period, and the proof of optimality follows the same reasoning. Since the response-time tests described above do not depend on any particular priority assignment, they can be applied to DM scheduling as well as RM scheduling.

2.4 Task interaction

The Liu and Layland model assumes that tasks are independent. In real systems that is not the case. There are resources, such as buffers and hardware

devices, that tasks must access in a mutually exclusive manner. The mechanisms that enforce mutual exclusion necessarily sometimes cause a task to *block* until another task releases a resource. For a task system to be feasible, the duration of such blocking must be bounded. Given such a bound, the feasibility analysis can be refined to take the worst-case effect of blocking into account.

In a fixed-priority preemptive scheduling system, blocking is called *priority inversion* because it results in a violation of the priority scheduling rule: if the highest priority task is blocked, a lower priority task will execute. Sha observed that the duration of priority inversion due to blocking may be arbitrarily long. For example, a high priority task may preempt a low priority task, the high priority task may then be blocked when it attempts to lock a resource already locked by the low priority task, and then the low priority task may be repeatedly preempted by intermediate priority tasks, so that it is not able to release the resource and unblock the high-priority task. Sha, Rajkumar, and Lehoczky introduced the family of *priority inheritance protocols* as a solution approach to the priority inversion problem [34]. The *priority inheritance protocol* (PIP) prescribes that if a higher priority task becomes blocked by a low priority task, the task that is causing the blocking should execute with a priority which is the maximum of its own nominal priority and the highest priority of the jobs that it is currently blocking.

However, the PIP does not prevent deadlocks. In addition, a job can be blocked multiple times. A solution is to add a new rule to the PIP: associate with each resource a priority, called the *priority ceiling* of the resource. The priority ceiling is an upper bound on the priority of any task that may lock the resource. A job may not enter its critical section unless its priority is higher than all the priority ceilings currently locked by other jobs. Under this ceiling rule, a job that may share resources currently locked by other tasks cannot enter its critical section. This prevents deadlocks. In addition, under the priority ceiling protocol a job can be blocked at most once. This notion of priority ceilings is the basis of several locking protocols, including the *priority ceiling protocol* (PCP) [30, 34], the *stack resource protocol* (SRP) [4], and the Ada 95 programming language ceiling locking protocol [5].

The feasibility analyses given in the previous section have been extended to account for the blocking that tasks may be subject to at run time. In the calculation of the feasibility of a task, its computation time is viewed as consisting of its worst-case execution time, the worst-case interference it can get from higher priority tasks, and the worst-case time for which it may be blocked.

Permitting tasks to communicate and synchronize via shared resources is only one form of task interaction that is not allowed within the Liu and Layland model. Another is precedence constraints or relations between tasks. When two tasks have a precedence relationship between them, the successor task cannot commence execution before the predecessor task has completed. While the choice of task offsets and periods can enforce precedence relations

implicitly, little work has appeared in the literature regarding the explicit analysis of precedence-related tasks within fixed-priority systems. In 1991, Harbour, Klein, and Lehoczky [19] considered tasks that are subdivided into precedence constrained parts, each with a separate priority.

2.5 Aperiodic tasks

In previous sections, we have discussed the scheduling of tasks with periodic releases and predictable execution times. However, many real-time systems also contain tasks whose processor demands do not fit that model. We use the term *non-periodic* for such tasks, whether they have significantly varying inter-release times, significantly varying execution times, no hard deadline, or some combination of these characteristics.

If the periodic task model is relaxed slightly, letting C_i be just the maximum execution time and letting T_i be the minimum inter-release time, Liu and Layland's analysis and most of the refinements that followed it remain valid [28]. However, for non-periodic tasks with large variations in inter-release times and/or execution times, reserving enough processor capacity to guarantee feasibility under this model is impractical.

One simple way to handle non-periodic tasks is to assign them priority levels below those of tasks with hard deadlines, i.e, relegate them to background processing. However, if there is more than one such task and they have quality-of-service requirements, average response-time requirements, or average throughput requirements, background processing is likely to be unsatisfactory.

Non-periodic tasks with quality-of-service requirements may be run at a higher priority level, under the control of a pseudo hard real-time server task such as a polling server [33]. A polling server is a periodic task with a fixed priority level (possibly the highest) and a fixed execution capacity. The capacity of the server is calculated off line and is normally set to the highest level that permits the feasibility of the hard-deadline periodic task set to be guaranteed. At run time, the polling server is released periodically. Its capacity is used to service non-periodic tasks. Once this capacity has been exhausted, execution of the polling server is suspended until the server's next (periodic) release. Since the polling server behaves like a periodic task, the feasibility analysis techniques developed for periodic tasks can be applied.

A polling server can guarantee hard deadlines for sporadic tasks, by appropriate choices of period and server budget, and can guarantee a minimum rate of progress for long-running tasks. It also is a significant improvement over background processing for aperiodic tasks. However, if aperiodic jobs arrive in a large enough burst to exceed the capacity of the server, then some of them must wait until its next release, leading to potentially long response times. Conversely, if no jobs are ready when the server is released, the high priority capacity reserved for it is wasted.

The Deferrable Server [41] and Sporadic Server algorithms [37] are based on principles similar to those underlying the polling server. However, they reduce wasted processor capacity by preserving it if there are no jobs pending when the server is released. Due to this property, they are termed *bandwidth preserving algorithms*. The two algorithms differ in the ways in which the capacity of the server is preserved and replenished and in the feasibility analysis needed to determine their maximum capacity. In general, both offer improved responsiveness over the polling approach. Even these more complex server algorithms are unable to make full use of the slack time that may be present due to the often favorable (i.e., not worst-case) phasing of periodic tasks. The Deferrable and Sporadic Server algorithms are also unable to reclaim spare capacity gained when, for example, other tasks require less than their worst-case execution time. This spare capacity can, however, be reclaimed by the Extended Priority Exchange algorithm [36]. Comparisons of these server schemes are provided by Bernat and Burns [9], who conclude that the best choice of server algorithm is application dependent.

Another approach to supporting soft-deadline tasks within a system that contains hard deadlines is to assign two priorities to the hard-deadline tasks [10]. When necessary, the task is promoted to the second, higher, priority to ensure that it will meet its deadline. In general, the soft tasks have deadlines below the second but above the initial deadline of these hard tasks.

Recently, Abdelzaher, Sharma, and Lu [1] have developed sufficient conditions for accepting firm deadline aperiodic work while guaranteeing that all deadlines of those tasks will be met. The admission algorithm requires knowledge of an arriving task's computation requirement and deadline. Given this knowledge, Abdelzaher derives a schedulability bound in the spirit of the Liu and Layland bound showing that if the workload level is kept below a certain synthetic utilization level, all admitted tasks will always meet their deadlines. Specifically, assume that each arriving aperiodic task has known computation time C and deadline D. At each instant of time, a *synthetic utilization* value, $U(t) = \sum \frac{C_i}{D_i}$, is computed where the sum extends over all tasks that have arrived and been accepted but whose deadlines have not yet expired. Abdelzaher et al. show that there is a fixed-priority policy that will meet all the deadlines of the accepted aperiodic tasks as long as the restriction $U(t) \leq 2 - \sqrt{2}$ is enforced. Arriving aperiodic tasks that would cause $U(t)$ to exceed this bound are not accepted for processing. Under a heavy workload, this algorithm will reject unschedulable requests and accept those whose deadline can still be guaranteed, leading to improved aperiodic utilization.

2.6 Overload management

Analyses of schedulability must make some assumptions about workload, which is ordinarily characterized by the worst-case execution time and the minimum inter-release time of each task. However, real task execution times may vary widely, and the actual worst-case execution time of a task may be

difficult or impossible to determine. Task release times may also vary from the ideal model. To cope with such variability, a system designer may try to (a) *prevent* overloads, by making safe assumptions about workload, which may be even more pessimistic than the actual worst case; (b) *tolerate* overloads, by providing some reduced but acceptable level of service when the workload exceeds normal expectations.

With fixed-priority scheduling, an execution time overrun (provided it is not in a non-preemptible section) or early task release may only delay the execution of lower priority tasks. If it is known which tasks may be released early or run over their nominal worst-case execution times, then only those tasks and the tasks of lower priority need to be designed to tolerate and recover from overruns.

Many practical systems, e.g., factory automation systems, PCs, and cell phones, have a mixture of real-time tasks and non-real-time tasks. A widely used practice is to give real-time tasks higher priorities. We know that real-time tasks can meet their deadlines as long as they are schedulable as a group. For example, in a modern PC, the audio and video will have higher priorities and typically have a total utilization that is far below the Liu and Layland bound. They can meet their deadlines, even if the system is overloaded by other activities such as compiling applications and backing up files. Such a simple but effective solution takes advantage of the nature of fixed-priority scheduling.

Even if all the real-time tasks are not schedulable under wost-case conditions, it may be possible to distinguish critical and non-critical tasks. That is, the system may be able to function tolerably for some period of time if all the critical tasks complete within their deadlines. Sha and Goodenough showed that it is easy to ensure that a set of critical tasks' deadlines will be met when non-critical tasks overrun their nominal worst-case executions times, provided the critical tasks never overrun their own nominal worst-case execution times and are schedulable as a group under those worst-case execution times [32]. The technique is to shorten the periods of the critical tasks (the period transformation mentioned in Section 2.1) so that the critical tasks all have higher RM priorities than the non-critical tasks. More precise control over the effects of overload may be achieved in other ways. Servers can be used to isolate the cause of overload or to provide resources for recovery. Another approach to overload is to define, if the application will allow, certain executions of each task to be "skippable" [8, 22].

If one designs to prevent overloads by sizing a system to function correctly under very conservative, safe assumptions about workload, there will ordinarily be significant spare capacity available at run time. This spare capacity becomes available for many reasons, including tasks completing in less than their worst-case execution time, sporadic tasks not arriving at their maximum rate, and periodic tasks not arriving in worst-case phasing. One then has the problem of *resource recovery*, that is, finding a way to use the spare capacity to enhance the quality of hard real-time services.

One important approach to designing to make good use of spare capacity, for both fixed-priority and dynamic-priority scheduling, is the *imprecise computation* model [35]. This is based on what are known more generally as "anytime algorithms" because they can provide a useful result any time they are queried or stopped. One such approach is to decompose every task τ_i into a *mandatory* subtask M_i and an *optional* subtask O_i. The mandatory subtask is the portion of the computation that must be done in order to produce a result of acceptable quality, whereas the optional subtask refines this result. Both subtasks have the same arrival time a_i and the same deadline d_i as the original task τ_i; however, O_i becomes ready for execution when M_i is completed.

3 Dynamic-Priority Scheduling

With dynamic-priority scheduling, task priorities are assigned to individual jobs. One of the most-used algorithms belonging to this class is the *Earliest Deadline First* (EDF) algorithm, according to which priorities assigned to tasks are inversely proportional to the absolute deadlines of the active jobs. The feasibility analysis of periodic task sets under EDF was first presented in 1973 by Liu and Layland [27], who showed that, under the same simplified assumptions used for RM scheduling, a set of n periodic tasks is schedulable by the EDF algorithm, if and only if

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1, \tag{4}$$

where C_i is the worst-case execution time of task τ_i and T_i is its period. In 1974, Dertouzos [16] showed that EDF is optimal among all preemptive scheduling algorithms, in the sense that, if there exists a feasible schedule for a task set, then the schedule produced by EDF is also feasible. Later, Mok presented another optimal algorithm, *Least Laxity First* (LLF) [29], which assigns the processor to the active task with the smallest laxity.[5] Although both LLF and EDF are optimal algorithms, LLF has a larger overhead due to the higher number of context switches caused by laxity changes at run time. For this reason, most of the work done in the real-time research community has concentrated on EDF to relax some simplistic assumption and extend the feasibility analysis to more general cases.

In this section we provide an overview of the key results related to EDF scheduling. We first present an analysis technique for verifying the feasibility of deadline-based schedules and briefly describe some efficient algorithms

[5] We recall that the laxity (or slack time) is the difference between the absolute deadline and the estimated worst-case finishing time.

for aperiodic task scheduling and shared resource management. Then, we describe some methods for dealing with overload conditions, and we conclude the section by presenting some open research issues.

3.1 Processor demand criterion

Under EDF, the analysis of periodic and sporadic tasks with deadlines less than periods can be performed by processor demand analysis, proposed in 1990 for strictly periodic tasks by Baruah, Rosier, and Howell [6], and for sporadic tasks by Baruah, Mok, and Rosier [7].

In general, the processor demand in an interval $[t_1, t_2]$ is the amount of processing time $g(t_1, t_2)$ requested by those jobs activated in $[t_1, t_2]$ that must be completed in $[t_1, t_2]$. Hence, the feasibility of a task set is guaranteed if and only if *in any interval of time* the total processor demand does not exceed the available time, that is, if and only if

$$\forall t_1, t_2 \quad g(t_1, t_2) \ \leq \ (t_2 - t_1).$$

Baruah, Rosier, and Howell showed that a set of periodic tasks simultaneously activated at time $t = 0$ is schedulable by EDF if and only if $U < 1$ and

$$\forall L > 0 \quad \sum_{i=1}^{n} \left\lfloor \frac{L + T_i - D_i}{T_i} \right\rfloor C_i \ \leq \ L. \tag{5}$$

Baruah, Mok, and Rosier showed that the points in which the test has to be performed correspond to those deadlines within the hyper-period H not exceeding the value $\max\{D_1, \ldots, D_n, L*\}$, where

$$L^* \ = \ \frac{\sum_{i=1}^{n} (T_i - D_i) U_i}{1 - U}.$$

It follows that when deadlines are less than or equal to periods, the exact feasibility analysis of EDF is of pseudo-polynomial complexity. Of course, when deadlines are equal to periods, the utilization test may be used, with a complexity of only $O(n)$.

3.2 Aperiodic task scheduling

The higher schedulability bound of dynamic scheduling schemes also allows achieving better responsiveness in aperiodic task handling. Several algorithms have been proposed in the real-time literature to handle aperiodic requests within periodic task systems scheduled by EDF [18, 38]. Some of them are extensions of fixed-priority servers, whereas some were directly designed for EDF.

The total bandwidth server approach

One of the most efficient techniques to safely schedule aperiodic requests under EDF is the total bandwidth server (TBS) [38], which assigns each aperiodic job a deadline in such a way that the overall aperiodic load never exceeds a specified maximum value U_s.

In particular, when the kth aperiodic request arrives at time $t = r_k$, it receives a deadline

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s},$$

where C_k is the execution time of the request and U_s is the server utilization factor (that is, its bandwidth). By definition $d_0 = 0$. Note that in the deadline assignment rule the bandwidth allocated to previous aperiodic requests is considered through the deadline d_{k-1}. Once the deadline is assigned, the request is inserted into the system ready queue and scheduled by EDF as any other periodic instance. As a consequence, the implementation overhead of this algorithm is practically negligible. The schedulability test for a set of periodic tasks scheduled by EDF in the presence of a TBS is given by the following theorem by Spuri and Buttazzo [38].

Theorem 1. *Given a set of n periodic tasks with processor utilization U_p and a TBS with processor utilization U_s, the whole set is schedulable by EDF if and only if*

$$U_p + U_s \leq 1.$$

Achieving optimality

The deadline assigned by the TBS can be shortened to minimize the response time of aperiodic requests, while maintaining the periodic tasks' schedulability. Buttazzo and Sensini [12] proved that setting the new deadline at the current estimated worst-case finishing time does not jeopardize schedulability.

The process of shortening the deadline can then be applied recursively to each new deadline until no further improvement is possible, given that the schedulability of the periodic task set must be preserved. If d_k^s is the deadline assigned to the aperiodic request J_k at the sth iteration and f_k^s is the corresponding finishing time in the current EDF schedule (achieved with d_k^s), the new deadline d_k^{s+1} is set equal to f_k^s. The algorithm stops either when $d_k^s = d_k^{s-1}$ or after a maximum number of steps defined by the system designer for bounding the complexity.

It is worth noticing that the overall complexity of the deadline assignment algorithm is $O(Nn)$, where N is the maximum number of steps performed by the algorithm to shorten the initial deadline assigned by the TBS. Finally, as far as the average case execution time of tasks is equal to the worst-case one, this method achieves optimality, yielding the minimum response time for each aperiodic task, as stated by the following theorem by Buttazzo and Sensini [12].

Theorem 2. *Let σ be a feasible schedule produced by EDF for a task set $\mathcal{T}$ and let f_k be the finishing time of an aperiodic request J_k, scheduled in σ with deadline d_k. If $f_k = d_k$, then $f_k = f_k^*$, where f_k^* is the minimum finishing time achievable by any other feasible schedule.*

Although the TBS can efficiently handle bursty sequences of jobs, it cannot be used to serve jobs with variable or unknown execution times. In this case, a budget management mechanism is essential to prevent execution overruns from jeopardizing the schedulability of hard tasks.

The constant bandwidth server

The constant bandwidth server (CBS) is a scheduling mechanism proposed by Abeni and Buttazzo [2] to implement resource reservations in EDF-based systems.

A CBS is characterized by a budget c_s, a dynamic server deadline d_s, and an ordered pair (Q_s, T_s), where Q_s is the maximum budget and T_s is the period of the server. The ratio $U_s = Q_s/T_s$ is denoted as the server bandwidth.

Each job served by the CBS is assigned a suitable deadline equal to the current server deadline, computed to keep its demand within the reserved bandwidth. As the job executes, the budget c_s is decreased by the same amount and, every time $c_s = 0$, the server budget is recharged to the maximum value Q_s and the server deadline is postponed by a period T_s to reduce the interference on the other tasks. Note that by postponing the deadline, the task remains eligible for execution. In this way, the CBS behaves as a work-conserving algorithm, exploiting the available slack in an efficient way, and providing better responsiveness as compared to non-work-conserving algorithms and to other reservation approaches that schedule the extra portions of jobs in background.

An important property is that, in any interval of time of length L, a CBS with bandwidth U_s will never demand more than $U_s L$, independently from the actual task requests. This property allows the CBS to be used as a bandwidth reservation strategy to allocate a fraction of the CPU time to soft tasks whose computation time cannot be easily bounded. The most important consequence of this result is that such tasks can be scheduled together with hard tasks without affecting the *a priori* guarantee, even in the case in which soft requests exceed the expected load.

3.3 Resource sharing

Under EDF, several resource access protocols have been proposed to bound blocking due to mutual exclusion, such as dynamic priority inheritance [40], dynamic deadline modification [20], and stack resource policy [4]. The stack resource policy is one of the most used methods under EDF for its properties and efficiency; hence it will be briefly recalled in the next section.

Stack resource policy

The stack resource policy (SRP) is a concurrency control protocol proposed by Baker [4] to bound the priority inversion phenomenon in static as well as dynamic-priority systems. In this work, Baker made the insightful observation that, under EDF, jobs with a long relative deadline can only delay, but cannot preempt, jobs with a short relative deadline. A direct consequence of this observation is that a job cannot block another job with longer relative deadline. Thus, in the study of blocking under EDF, we only need to consider the case where jobs with longer relative deadlines block jobs with shorter relative deadlines.

This observation allows Baker to define preemption levels separately from priority levels under EDF, in such a way that they are inversely proportional to relative deadlines. It follows that the semaphore preemption ceiling under SRP can be defined in the same way as the priority ceiling under PCP.

Hence, under SRP with EDF, each job of τ_i is assigned a priority p_i according to its absolute deadline d_i and a static *preemption level* π_i inversely proportional to its relative deadline. Each shared resource is assigned a ceiling which is the maximum preemption level of all the tasks that will lock this resource. Moreover, a *system ceiling* Π is defined as the highest ceiling of all resources currently locked.

Finally, the SRP scheduling rule requires that

> *"a job J is not allowed to start executing until its priority is the highest among the active tasks and its preemption level is greater than the system ceiling* Π*".*

SRP guarantees that, once a job is started, it will never block until completion; it can only be preempted by higher priority tasks. SRP prevents deadlocks, and a job can be blocked for at most the duration of one critical section.

Under the SRP there is no need to implement waiting queues. In fact, a task never blocks during execution: it simply cannot start executing if its preemption level is not high enough. As a consequence, the blocking time B_i considered in the schedulability analysis refers to the time for which task τ_i is kept in the ready queue by the preemption test.

The feasibility of a task set with resource constraints (when only periodic and sporadic tasks are considered) can be tested by the following sufficient condition [4]:

$$\forall i,\ 1 \le i \le n \quad \sum_{k=1}^{i} \frac{C_k}{T_k} + \frac{B_i}{T_i} \ \le\ 1, \tag{6}$$

where B_i is the maximum blocking time of task τ_i and it is assumed that all the tasks are sorted by decreasing preemption levels, so that $\pi_i \ge \pi_j$ only if $i < j$.

As a final remark, the SRP allows tasks to share a common run-time stack, allowing large memory savings if there are many more tasks than relative priority levels.

Under EDF, resources can also be shared between hard tasks and soft tasks handled by an aperiodic server. Ghazalie and Baker [18] proposed to reserve an extra budget to the aperiodic server for synchronization purposes and used the utilization-based test [27] for verifying the feasibility of the schedule. Lipari and Buttazzo [26] extended the analysis to a TBS. Caccamo and Sha [15] proposed another method for capacity-based servers, like the CBS, which also handles soft real-time requests with a variable or unknown execution behavior. The method is based on the concepts of *dynamic preemption levels* and allows resource sharing between hard and soft tasks without jeopardizing the hard tasks' guarantee.

3.4 Overload management

An overload condition is a critical situation in which the computational demand requested by the task set exceeds the time available on the processor, so that not all tasks can complete within their deadlines. When a task executes more than expected, it is said to overrun. This condition is usually transient and may occur either because jobs arrive more frequently than expected (*activation overrun*) or because computation times exceed their expected value (*execution overrun*). A permanent overload condition may occur in a periodic task system, when the total processor utilization exceeds one. This could happen after the activation of a new periodic task, or if some tasks increase their activation rate to react to some change in the environment. In such a situation, computational activities start to accumulate in the system's queues, and the task response times tend to increase indefinitely.

Transient overloads

Transient light overloads due to activation overruns can be safely handled by an aperiodic server that, in the case of a bursty arrival sequence, distributes the load more evenly according to the bandwidth allocated to it. In heavier load conditions, admission control schemes may be necessary to keep the load below a desired threshold [39].

Overloads due to execution overruns can be handled through a resource reservation approach [2]. The idea behind resource reservation is to allow each task to request a fraction of the available resources, just enough to satisfy its timing constraints. The kernel, however, must prevent each task from consuming more than the requested amount, to protect the other tasks in the systems (temporal protection). In this way, a task receiving a fraction U_i of the total processor bandwidth behaves as if it were executing alone on a slower processor with a speed equal to U_i times the full speed. The advantage of this method is that each task can be guaranteed in isolation, independently of the

behavior of the other tasks. A simple and effective mechanism for implementing temporal protection under EDF is the CBS [2]. To properly implement temporal protection, however, each task τ_i with variable computation time should be handled by a dedicated CBS with bandwidth U_{s_i}, so that it cannot interfere with the rest of the tasks for more than U_{s_i}.

Although resource reservation is essential for achieving predictability in the presence of tasks with variable execution times, the overall system's performance becomes quite dependent on a correct resource allocation. For example, if the CPU bandwidth allocated to a task is much less than its average requested value, the task may slow down too much, degrading the system's performance. On the other hand, if the allocated bandwidth is much greater than the actual needs, the system will run with low efficiency, wasting the available resources.

Two reclaiming techniques have been proposed to cope with an incorrect bandwidth assignment. The *CApacity SHaring* (CASH) algorithm [14] works in conjunction with the CBS. Its main idea can be summarized as follows: 1) whenever a task completes its execution, the residual capacity (if any) is inserted with its deadline in a global queue of available capacities, the CASH queue, ordered by deadline; 2) whenever a new task instance is scheduled for execution, the server tries to use the residual capacities with deadlines less than or equal to the one assigned to the served instance; if these capacities are exhausted and the instance is not completed, the server starts using its own capacity. The main benefit of this reclaiming mechanism is to reduce the number of deadline shifts in the CBS, thereby enhancing the responsiveness of aperiodic tasks.

The *Greedy Reclamation of Unused Bandwidth* (GRUB) [25] algorithm is another server-based technique to reclaim unused processor bandwidth. According to the GRUB algorithm, each task is executed by a distinct server S_i, where each server is characterized by two parameters: a *processor share* U_i, and a *period* P_i. GRUB is able to emulate a virtual processor of capacity U_i when executing a given job, and it uses a notion of virtual time to enforce isolation among different applications and to safely reclaim all unused capacities generated by periodic and aperiodic activities. Compared to CASH, the GRUB algorithm has more overhead due to its virtual time management, but it achieves better performance in terms of aperiodic responsiveness.

Permanent overloads

Permanent overload conditions in periodic task systems can be handled using three basic approaches that allow keeping the load below a desired value. A first method reduces the total load by properly skipping (i.e., aborting) some jobs in the periodic tasks, in such a way that a minimum number of jobs per task are guaranteed to execute within their timing constraints. In a second approach, the load is reduced by enlarging task periods to suitable values, so that the total workload can be kept below a desired threshold. In the third

approach, the load is reduced by decreasing the computational requirements of the tasks, trading quality of results with predictability.

Permitting skips in periodic tasks increases system flexibility, since it allows making better use of resources and scheduling systems that would otherwise be overloaded. The *job skipping* model was originally proposed by Koren and Shasha [22], who showed that making optimal use of skips is NP-hard and presented two algorithms that exploit skips to increase the feasible periodic load and schedule slightly overloaded systems.

According to the job skipping model, the maximum number of skips for each task is controlled by a specific parameter associated with the task. In particular, each periodic task τ_i is characterized by a worst-case computation time C_i, a period T_i, a relative deadline equal to its period, and a skip parameter S_i, $2 \leq S_i \leq \infty$, which gives the minimum distance between two consecutive skips. For example, if $S_i = 5$, the task can skip one instance of every five. When $S_i = \infty$, no skips are allowed and τ_i is equivalent to a hard periodic task. The skip parameter can be viewed as a *quality-of-service* (QoS) metric (the higher the value of S, the better the quality of service). In the same work, Koren and Shasha provided a sufficient condition for guaranteeing a set of skippable periodic tasks under EDF. It is worth noting that, if skips are permitted in the periodic task set, the spare time saved by rejecting the skipped instances can be reallocated for other purposes. In [13], Caccamo and Buttazzo generalized the results in [22] by identifying the amount of bandwidth savings achieved by skips to advance the execution of aperiodic tasks.

A second approach, named *elastic scheduling*, is able to handle permanent overloads by varying task periods. Whenever the total processor utilization is greater than one, the utilization of each task needs to be reduced so that the total utilization becomes equal to a desired value $U_d \leq 1$. This can be done as in a linear spring system, where springs are compressed by a force F (depending on their elasticity) up to a desired total length. To apply this method, each task needs to be specified with additional parameters, including a range of periods and an elastic coefficient, used to diversify compression among tasks, so that utilization reduction is higher for tasks with higher elasticity. As shown in [11], in the absence of period constraints, the utilization U_i of each compressed task can be computed in order $O(n)$ (where n is the number of tasks), whereas in the presence of period constraints ($T_i \leq T_{i_{max}}$), the problem of finding the U_i values requires an iterative solution of $O(n^2)$ complexity. The same algorithm can be used to reduce the periods when the overload is over, thus adapting task rates to the current load condition to better exploit the computational resources. Another method is the imprecise computation that was discussed in Section 2.6.

4 Challenges Ahead

We have reviewed some of the major real-time scheduling results. Looking at the "big picture," we have witnessed the changing trends in real-time systems. System architecture has changed from a federated system architecture to an integrated system architecture and then to a system of systems. Each shift has brought about enormous challenges to the available technological infrastructure.

In a federated architecture, a system is characterized by a collection of private hardware resources, a small number of high volume, high variability sensor data streams on dedicated links, loosely coupled distributed actions, and hardware-based isolation and protection. Under this type of architecture, the task of managing shared resources focuses mainly on how to handle periodic data flow, driven by signal processing and control. The existing real-time computing infrastructure has served most practitioners well for this type of application.

In an integrated system architecture, sensors, communication channels, and processors are extensively shared. The large number of possible configurations becomes a challenge. The current generation of schedulability analysis tools offers inadequate support for system architects. They must manually create the alternative options and then check the schedulability of each option. It would be preferable to have tools that automatically search the design space and perform sensitivity analysis taking into account the uncertainty of task parameters. During the system engineering phase, the values of task parameters are often educated guesses. Finally, from the perspective of run-time reconfiguration, dynamic-priority scheduling theory offers several advantages. In addition to the potential of higher schedulability, the feasibility analysis of dynamic priority scheduling is often faster. We are looking forward to the maturing and subsequent use of dynamic scheduling theory in practice.

In modern integrated systems, there are substantial high volume and high variability imaging data streams. Depending on the application, these streams can have either hard or soft end-to-end deadlines. If the images are used for steering a vehicle, they will have a hard end-to-end deadline and tight jitter tolerance. Such hard-deadline streams pose challenges to traditional scheduling theory using worst-case assumptions. The large execution time variability causes inefficiency. Many algorithms have been developed to capture the unused cycles to improve local aperiodic response times. How to effectively use such transient surpluses to improve end-to-end responses requires more study. The soft real-time image data streams also pose challenges to the statistical approach. In fact, the processing times of an imaging data stream at the various nodes on its path are positively correlated, not independent. In addition, on congested resources such as shared buses, an image data stream can consume a significant fraction of a shared resource, instead of consuming a small fraction of a shared resource that allows the use of the law of large numbers.

A system of systems is often a large distributed system, where keeping distributed views and actions timely and consistent is at the heart of collaborative actions. Ideally, we would like to keep distributed views, state transitions, and actions consistent with one another. In business systems, the consistency of a distributed system is managed by atomic operations. Simply put, atomic operations wait for every working component to be ready and then commit the operations. However, this may not be viable for real-time systems. Metaphorically speaking, the train must leave the station without waiting for every passenger to board. However, those components that are left behind with outdated views and states must quickly catch up and re-synchronize themselves with the system. If more and more components fall out of synchronization, the system of systems will fall apart. Handling the interactions between timing constraints, consistency requirements, and re-synchronization in a large distributed system remains a challenge. As a networked embedded system of systems grows larger and its coordination becomes tighter, so grows the importance of this technological challenge. The re-synchronization loop is a form of feedback control: feedback is a powerful technique which has yet to be fully exploited in the control of the behavior of computing systems in the face of uncertainty.

Another characteristic of a system of systems is that a wide variety of real-time, fault tolerance, and security protocols are used in different systems, because most of the systems of systems are integrated, not built from scratch. It is well known that perfectly fine medicines when taken alone can react pathologically when taken together. Priority inversion is an example of a pathological interaction between an independently developed synchronization protocol and a priority scheduling protocol. This is not an easy problem to solve because the scope of modern technologies is so large and complex. To advance any area, one must specialize. As a result, we have specialized real-time, fault tolerance, and security communities focusing on improving the results in one dimension with little attention on how separately developed protocols may interact. We need to create a forum for the co-development/integration of real-time, fault tolerant, security, communication, and control protocols. Research is needed to formally verify that protocols do not invalidate each others' pre-conditions when they interact.

Looking ahead, much remains to be done in the creation of a new real-time computing infrastructure for modern real-time systems. It will be an exciting time!

References

1. T. Abdelzaher, V. Sharma, and C. Lu. A utilization bound for aperiodic tasks and priority driven scheduling. *IEEE Trans. on Computers*, 53(3):334–350, Mar. 2004.

2. L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. 19th IEEE Real-Time Systems Symposium*, Madrid, Spain, Dec. 1998.
3. N. C. Audsley, A. Burns, R. Davis, K. Tindell, and A. J. Wellings. Fixed priority preemptive scheduling: A historical perspective. *Real Time Systems*, 8(3):173–198, 1995.
4. T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–100, Mar. 1991.
5. T. P. Baker and O. Pazy. Real-time features for Ada 9X. In *Proc. 12th IEEE Real-Time Systems Symposium*, pages 172–180, 1991.
6. S. K. Baruah, R. R. Howell, and L. E. Rosier. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *Real-Time Systems*, 2:173–179, 1990.
7. S. K. Baruah, A. K. Mok, and L. E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. *Proc. 11th IEE Real-Time Systems Symposium*, pages 182–190, 1990.
8. G. Bernat and A. Burns. Combining (n, m)-hard deadlines with dual priority scheduling. In *Proc. 18th IEEE Real-Time Systems Symposium*, pages 46–57, 1997.
9. G. Bernat and A. Burns. New results on fixed priority aperiodic servers. In *Proc. 20th IEEE Real-Time Systems Symposium*, pages 68–78, 1999.
10. A. Burns and A. J. Wellings. Dual priority assignment: A practical method of increasing processor utilization. In *Proc. Fifth Euromicro Workshop on Real-Time Systems*, pages 48–53, Oulu, Finland, 1993. IEEE Computer Society.
11. G. Buttazzo, G. Lipari, M. Caccamo, and L. Abeni. Elastic scheduling for flexible workload management. *IEEE Trans. on Computers*, 51(3):289–302, Mar. 2002.
12. G. Buttazzo and F. Sensini. Optimal deadline assignment for scheduling soft aperiodic tasks in hard real-time environments. *IEEE Trans. on Computers*, 48(10):1035–1052, Oct. 1999.
13. M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *Proc. IEEE 18th Real-Time Systems Symposium*, pages 330–339, San Francisco, 1997.
14. M. Caccamo, G. Buttazzo, and L. Sha. Capacity sharing for overrun control. In *Proc. 21st IEEE Real-Time Systems Symposium*, pages 295–304, Orlando, FL, USA, Dec. 2000.
15. M. Caccamo and L. Sha. Aperiodic servers with resource constraints. In *Proc. 22nd IEEE Real-Time Systems Symposium*, pages 161–170, London, UK, Dec. 2001.
16. M. L. Dertouzos. Control robotics: The procedural control of physical processes. *Information Processing*, 74, 1974.
17. L. Doyle and J. Elzey. Successful use of rate monotonic theory on a formidable real time system. In *Proc. 11th IEEE Workshop on Real-Time Operating Systems and Software*, pages 74–78, May 1994.
18. T. M. Ghazalie and T. P. Baker. Aperiodic servers in a deadline scheduling environment. *Real-Time Systems*, 9:31–67, 1995.
19. M. G. Harbour, M. H. Klein, and J. P. Lehoczky. Fixed priority scheduling of periodic tasks with varying execution priority. In *Proc. 12th IEEE Real-Time Systems Symposium*, 1991.

20. K. Jeffay. Scheduling sporadic tasks with shared resources in hard-real-time systems. In *Proc. 13th IEEE Real-Time Systems Symposium*, pages 89–99, Phoenix, AZ, USA, Dec. 1992.
21. M. Joseph and P. Pandya. Finding response times in a real-time system. *BCS Computer Journal*, 29(5):390–395, 1986.
22. G. Koren and D. Shasha. Skip-over: Algorithms and complexity for overloaded systems that allow skips. In *Proc. IEEE Real Time System Symposium*, pages 110–117, Pisa, 1995.
23. J. P. Lehoczky, L. Sha, and D. Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proc. 10th IEEE Real-Time Systems Symposium*, pages 166–171, 1989.
24. J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation (Netherlands)*, 2(4):237–250, 1982.
25. G. Lipari and S. K. Baruah. Greedy reclamation of unused bandwidth in constant bandwidth servers. In *Proc. 12th Euromicro Conference on Real-Time Systems*, pages 192–200, Stockholm, Sweden, June 2000.
26. G. Lipari and G. Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of Systems Architecture*, 46(4):327–338, Jan. 2000.
27. C. L. Liu and J. W. Layland. Scheduling alghorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, Jan. 1973.
28. J. W. S. Liu. *Real-Time Systems.* Prentice-Hall, Upper Saddle River, NJ, 2000.
29. A. K. Mok. *Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment.* Ph.D. thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, Cambridge, MA, 1983.
30. R. Rajkumar, L. Sha, and J. P. Lehoczky. Real-time synchronization protocols for multiprocessors. In *Proc. 9th IEEE Real-Time Systems Symposium*, pages 259–269, 1988.
31. L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. Mok. Real time scheduling theory: A historical perspective. *Real-Time Systems*, 28(2–3):101–155, Nov. 2004.
32. L. Sha and J. Goodenough. Real-time scheduling theory and Ada. *IEEE Computer*, 23(4):53–62, 1990.
33. L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for some practical problems in prioritizing preemptive scheduling. In *Proc. 7th IEEE Real-Time Sytems Symposium*, pages 181–191, 1986.
34. L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronisation. *IEEE Trans. on Computers*, 39(9):1175–1185, 1990.
35. W. Shih, W. S. Liu, and J. Chung. Algorithms for scheduling imprecise computations with timing constraints. *SIAM Journal of Computing*, 20(3):537–552, July 1991.
36. B. Sprunt, J. Lehoczky, and L. Sha. Exploiting unused periodic time for aperiodic service using the extended priority exchange algorithm. In *Proc. 9th IEEE Real-Time Systems Symposium*, pages 251–258, 1988.
37. B. Sprunt, L. Sha, and L. Lehoczky. Aperiodic task scheduling for hard real-time systems. *Real-Time Systems*, 1(1):27–60, 1989.

38. M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, Mar. 1996.
39. J. A. Stankovic, C. Lu, S. Son, and G. Tao. The case for feedback control real-time scheduling. In *IEEE Proc. 11th Euromicro Conference on Real-Time Systems*, pages 11–20, York, U.K., June 1999.
40. J. A. Stankovic, K. Ramamritham, M. Spuri, and G. Buttazzo. *Deadline scheduling for real-time systems.* Kluwer, Boston-Dordrecht-London, 1998.
41. J. Strosnider, J. P. Lehoczky, and L. Sha. The deferrable server algorithm for enhanced aperiodic responsiveness in real-time environments. *IEEE Trans. on Computers*, 44(1):73–91, Jan. 1995.

Network Fundamentals

David M. Auslander[1] and Jean-Dominique Decotignie[2]

[1] Mechanical Engineering Department, University of California, Berkeley, CA 94720-1740, U.S.A. dma@me.berkeley.edu
[2] Centre Suisse d'Electronique et de Microtechnique, Jaquet-Droz 1, 2000 Neuchatel, Switzerland jean-dominique.decotignie@csem.ch

1 Networking for Control Systems

1.1 Benefits of networking in control systems

Networks are the first widely applicable means of including multiple cooperating computers in the same control system. There are two primary motivations for wanting multiple computers in a control system: additional computing power and distribution of computing to match the target system's topology. While more computing power is always a good thing, in control systems computing power is valued according to priority. What is most valuable is high priority computing power. Given the sequential architecture of computers, only one activity can occupy the highest priority slot. With several computers in the system, there can be a highest priority activity on each of them.

The cost structure of computing also motivates the desire for obtaining added computing power via several computers. At any given technological time point, going beyond a certain computing capability in a single processor becomes very expensive. If more computing power than that limit is needed, the most cost-effective way to get it is with multiple processors.

System topology issues can be as strong or even stronger than computing power in motivating multiple computer control systems. Here the motivation is data and information integrity and reduced cabling. Traditional control systems relied on analog information transmission, both for sensing and for actuation. Analog cables carry one signal per cable. Analog signals are also susceptible to contamination from a variety of noise sources. Even when modulation schemes are used, significant noise susceptibility remains. These attributes of analog systems provide a powerful incentive for the use of networking in control. This incentive has both operational and capital cost aspects.

Operationally, digital signals can have arbitrarily strong protection against noise. Furthermore, the cost of that protection scales reasonably with the severity of the noise, whereas beyond certain practical limits, the costs of

protecting analog signals escalates rapidly. Thus, with proper design, once in digital format, a very high degree of data integrity can be ensured.

Overall system cost is another major motivation for networked control systems. In systems of even modest complexity, cabling is a major system cost and is also a major source of reliability problems. Well-designed networked control systems can dramatically reduce the amount of cabling—unlike analog cabling, a digital cable can carry multiple signals on a single channel.

1.2 Costs of networking in control systems

Economists warn us that there is "no such thing as a free lunch," so we should expect some cost to go with these benefits. The costs are economic—the infrastructure of the network, and operational—the computational overhead used to run the network, the added complexity of doing control over a network, and particular network characteristics that impact on control system performance.

The functional costs associated with the use of networks in control systems come from network performance specifications. The most common network specifications come from the larger business and personal computing environments. In these cases, the most important network attribute is throughput —the amount of information that can be passed through the network per unit of time. This is usually expressed in bits per second (bps) and is most properly measured as the average rate for moving a large amount of information, for example, downloading a large file. Control systems, however, operate on short time scales. The most important property of a network in a control system is the delay or latency time: How long does it take for information generated on one computer to be available for use on another? When the network is used as part of a feedback control loop, these delays degrade loop performance and can cause instability. When the network is only being used at higher levels, delays cause errors in synchronizing different parts of the system. It is true that networks with higher throughput will often have shorter latency times than networks with lower throughput, but the latency times depend on many other properties as well.

The overhead associated with networking becomes critical when multiple computers are being used to augment computing power in a control system. The overhead is incurred in preparing information for network transmission, transferring the information to the network interface so it can be transmitted on the network, retrieving information from the network interface, and decoding the received information for use by the program. Because of this overhead, the gain in computing power is never proportional to the number of added computers. It can, in fact, become negative as the overhead eats up all of the added computing power of the extra computers!

1.3 Time and event triggering in control systems

Control systems are driven by events. When specified events occur, the control system must take some action within a specified time (latency). One particular class of events, those based on time, are so ubiquitous in control systems that they are usually classified separately. Certainly, the single most common trigger for control system activity is the sample time of a discrete-time feedback loop.

In time-triggered (TT) systems all activities must be known in advance and thus it is easier to predict performances. The price to pay is inflexibility, which means that unpredicted events will not be treated. A second characteristic of TT systems is that the time at which every computation is made (and every message is transferred) is known beforehand. In fact, everything runs as if we were in the worst case with respect to overlap of event timings. In contrast, in event-triggered (ET) systems, some properties may be predicted but they are not deterministic, for example, the timing for some specific event. There is more flexibility in handling unforeseen cases and under normal circumstances, the performance is better than that of an "equivalent" TT system (we do not make the worst-case assumption). However, such systems usually require a substantial safety margin in their capabilities (processor speed, network bandwidth, memory size, etc.) so that unlikely but possible circumstances will not cause dangerous system errors.

Networks pose particular problems when it comes to triggering of control system events. In addition to the delay issues noted above, depending on the type of network, the delay times might have stochastic variations. Even the concept of time must be reexamined. In a single-processor control system, "time" is based on a clock that is part of the processor's hardware. With multiple computers, each one has its own clock. These clocks will drift with respect to each other, so relative timing of actions in the control system cannot be guaranteed unless specific measures are taken to keep them synchronized.

An even more insidious problem can occur when event-triggering information is sent across a network. Depending on the network type and configuration, the order of event notifications at the target node can be different than the order in which the events occurred. Thus, any actions that are dependent on event order can trigger incorrectly.

1.4 Network design for control

The advantages of networking in control systems are so strong that any systems above some minimal level of complexity are likely to utilize networking. However, networking in control has operational requirements that are quite different from networking in a general computing environment. Not only is the selection of network type important, but the design and configuration of the network are crucial for achieving satisfactory performance.

2 What Is a Network?

2.1 Network basics

To be a "network" in the computational sense a configuration of computers must:

- Share access to a common medium
- Encapsulate information into packets
- Share a common understanding of the meaning of the information.

A computer on a network, a network "node," can use the shared medium to access any other computer on the network. Packetizing the information to be exchanged allows for efficient use of the network resources. In fact, packetizing is why the modern network has achieved such incredible impact. Very large shared-access media have existed before—think of the pre-modern telephone network (say, before 1970), but this system was very expensive to use because it provided a dedicated point-to-point connection in order to complete a call. The even earlier party-line telephone networks made the shared access totally obvious to the user.

Although people usually use the telephone network as a prototypical example of a network, because of its point-to-point nature it is missing one of the key ingredients—packetizing. Surprisingly, a better example is the postal system (yes, snail mail!). Postal systems have provided low-cost, reliable communication for several hundred years. The key to the effectiveness of the postal system is the envelope. It encapsulates the information in a way that is independent of the nature of the information being transmitted. The envelope also includes addressing information in a standard format, sender identification, and payment information. In computer networking this is called "header" information. The very notion of "packet" as used in computer networking comes from postal mail. Here are a couple of dictionary definitions (via dictionary.com):

- ...a vessel employed by government to convey dispatches or mails (*Webster's Revised Unabridged Dictionary*, ©1996, 1998 MICRA, Inc.)
- a collection of things wrapped or boxed together (WordNet ®1.6, ©1997 Princeton University).

2.2 Local and wide area networks

A local area network (LAN) uses a single, common medium to connect a set of computer nodes. All of the nodes are energetically connected to each other (the form of energy depends on the medium—electrical, optical, radio frequency, etc.). As the name implies, LANs cover relatively compact areas. A primary issue in LANs is controlling access to the medium by each of the nodes in order to maintain order, make efficient use of the medium, and to

establish some notion of "equity," that is, making sure each computer node has appropriate access to the network.

A wide area network (WAN) is made up by interconnecting LANs. The connection is accomplished by converting the information flowing in physical form on the LAN to information (as data stored in a computer) and then retransmitting it to establish connections to computers not on the local network. The computers that accomplish the translation and retransmission are called "routers." The most prominent example of a WAN is the Internet. In the course of traversing a WAN, the information is typically carried over several different physical media along the way.

There are profound differences in LANs and WANs vis-á-vis control applications. Although many networking technologies are equally applicable to LANs or WANs, the performance issues are very different. Some performance factors that can differ significantly in LANs and WANs are:

- Information delay time
- Network traffic control
- Lost packets
- Packet ordering
- Security.

Once a packet crosses the boundary established by a router, "it's a big bad world out there!" A LAN can be physically secured if all of the media used in the network are on the same premises as the control system. Within the LAN all of the computer nodes are in physical contact with one another. Packets that are routed onto the WAN share traffic with all other users of the segments that the packets traverse. Because of that, performance at the level needed for control of physical systems can become quite unpredictable. The time it takes to deliver a packet depends on the route the packet takes and how heavily loaded those links are. While traffic on a LAN can be regulated, other factors affect load on WANs. Packets in the wide area environment can be lost or, because of routing differences, can arrive out of order. Security is likewise a much more difficult problem when packets move through a WAN.

The general design rule is thus that time critical operations in networked control systems are usually confined to well-designed LANs. Wide area usage in control is usually limited to obtaining diagnostic information, very high level supervisory access, etc.

2.3 Layering

Layering in network technology separates functional elements of a network into layers and then very carefully defines how each layer interacts with the layers above and below it. This model has successfully allowed specialists at each layer to refine that layer's technology while, because of the strongly defined layer interfaces, providing successful interoperability across systems from various manufacturers.

The classic work on layered design of networks is the Open Systems Interconnection (OSI) model from the International Organization for Standardization (ISO, know colloquially in English as the International Standardization Organization, http://www.iso.org/iso/en/ISOOnline.openerpage). Although this model is not used directly in commercial networking software, it forms the basis for design and construction of many networking technologies.

The OSI model has seven layers ([1], [3], p. 28):

1. *Physical layer.*
   Defines the physical and electrical characteristics of the network.
2. *Data link layer.*
   Defines the strategy for sharing the physical medium.
3. *Network layer.*
   Provides a means for communicating among open systems to establish, maintain and terminate network connections.
4. *Transport layer.*
   Ensures data reliability and integrity to the Session Layer (layer 5).
5. *Session layer.*
   Provides for two communicating entities to exchange data with each other.
6. *Presentation layer.*
   This is where application data is either packed or unpacked, ready for use by the running application.
7. *Application Layer.*
   This layer is for end-user and end-application protocols.

The top and bottom layers are quite intuitive. The physical layer is obvious in the network wires, connectors, etc. The application layer manifests itself in the many everyday applications that are network based, e-mail, for one. The middle layers, however, are largely invisible to most computer network users. Layer 3, for example, the network layer, is where the routers (mentioned above in connection with WANs) operate.

By far the most widely used network technology is the Transmission Control Protocol operating over the Internet Protocol (TCP/IP). TCP/IP is based on a four-layer model that was largely designed before the final release of the OSI model. The TCP/IP model consists of ([4], [2], [3]):

1. Link layer
2. Network layer
3. Transport layer
4. Application layer

Other than aggregating network functionality more than the OSI 7 layer model, the TCP/IP layers provide the same technology modularization that the OSI model does. The success of the layered model can be measured by the ease with which technology from many suppliers can be intermixed in the same network. As with any modular scheme, however, a major limitation

is that any optimization that crosses layers cannot be implemented without breaking the layered model.

These modules, which supply basic network functionality, are called "layers" in network technology because information goes through them in sequence. Information that starts in an application goes down through the layers in the originating computer until it hits the lowest level, where it enters the network and is transmitted to the target computer. The information then goes up through the layers until it gets to the application on the target computer that needs the data. Implementations of layered network systems are generally called "stacks" for that reason.

The full layered model is only required for WANs. Not all of the middle layers are required for LANs because each node on a local network is physically connected to all other nodes and so sees all traffic on the network. It only reacts to packets that are intended for it, ignoring the others. A network technology intended only for local area use can thus be simpler than one intended for both local and wide area use.

2.4 Shared access

Because network media are shared, there must be a mechanism to ensure that all participating nodes have "equitable" access. Equitable is in quotes because it does not necessarily mean equal as might be implied. Network technologies can be designed "flat," that is, where equitable does mean equal, or they can be designed with varying classes of service or priorities. In control systems, it may be important to establish priorities in order to make sure that the most critical activities get preference.

The nature of a shared medium is that all nodes of the local network can listen simultaneously, but only one at a time can be transmitting. The challenge, then, is to figure out a method that gives all nodes on the network appropriate access for transmitting packets.

There are two primary methods of controlling access to the network medium by a computer node: token passing and collision detection. There are also a number of variants that combine properties from both of these. A good way to visualize these methods is by analogy to conversations among people. The shared medium is the audio environment for verbal communication among them. For the equivalent of token passing, imagine a group of people sitting around a table. Each one gets to speak in turn; when a person is done, the person to the right (or left) gets a turn. The person speaking has the "token" giving him/her permission to speak. Sometimes, an actual object is used for the token to give people a more concrete visualization and keep a meeting more orderly. Once a person starts speaking, there is often a rule on how long that person can speak before passing the token to the next person.

A parliamentary proceeding, or one using Robert's Rules of Order, is a variant on a token passing situation—token passing with signaling. The meeting chair is equivalent to a network master. The token is passed to whomever raises

a hand (or some other signaling mechanism). Signaling on a network requires additional communication channels (in the human situation, vision is usually used for signaling). Because networks are designed for efficient long-distance communication, it is not normally economic to include signaling channels.

A group of people in normal conversation use collision detection to manage access to the shared audio channel. If a person wishes to speak he/she first listens to see if anyone else is speaking. If not, the person wishing to speak starts speaking. At the same time, he/she listens—the speaker continues if nothing is heard but his/her own voice. If other voices are mixed with it, then it means that someone else is starting to speak simultaneously. If there is a conflict, both speakers stop. After a short period one or the other (or both!) will try again. This is a collision detection method of granting access. It is simpler to administer than a token ring, but the performance is statistical in nature and deteriorates rapidly at high load.

Face-to-face conversation as described above also has elements of signaling since there are visual cues as to who would like to speak in addition to just the audio collisions. A purer form of collision detection occurs in a teleconference. There, the only cues are audio, so no added signaling channels are present.

A strong argument can be made that a token ring architecture, with its deterministic behavior and better performance at high load (relative to capacity), would lead to its choice over collision detection as the mainstream network architecture. That has not happened, however. By far the dominant mainstream network architecture is to use Ethernet as the physical layer for TCP/IP. Ethernet uses collision detection for access control (carrier sense multiple access/collision detection, CSMA/CD).

The argument is even stronger for control systems where network performance can be a critical factor in maintaining acceptable control of target physical systems. While some dedicated control system network architectures do use token ring or similar access methods, the enormous international investment in Ethernet technology makes it very hard to beat in either cost or performance—standard business-use computers are now being delivered with gigabit/sec Ethernet connections. Maintenance of predictable performance within the statistical universe of collision detection, however, requires that the network load be kept "light." A rule-of-thumb for "light" is 40% or less of maximum throughput rate. Thus, a collision detection network used in a control system where predictable performance is essential must be derated to somewhat less than half its stated rate.

A number of network architectures solve the access problems with combinations of these techniques. For example, the Controller Area Network (CAN) uses collision detection combined with bit arbitration ([14]). A variant of CAN, called CANopen, and Profibus, for example, use a master-slave architecture (also called primary-secondary) in which one node controls all communication on the network. Each architecture carries certain benefits and costs; these will be discussed below for some of the more popular network technologies.

This argument continues in the control system world. There are successful implementations of standard TCP/IP over Ethernet, there are a number of proposals for one form or another of industrial Ethernet, and there are many control-specific network architectures.

2.5 Protocols

A protocol is a well-defined set of rules for interactions between entities. In network systems, protocols operate at each network layer. The power of the system is that in using a protocol at any given network layer only the interaction with the partner network node need be considered—from a functional point of view, the other layers are completely transparent. The *performance* will depend entirely on how the other layers are implemented but not the function.

Using networking for control systems may require using existing protocols or writing new ones. Using existing protocols is very much like using any other software package. Software to utilize the protocols is written to utilize the application programming interface (API). Writing new protocols is considerably more complex. Most network protocols written for control systems are at the highest layer—the application layer. An application layer protocol makes use of the already defined protocols in the layer below (for TCP/IP, for example, that would be the transport layer). Designing a protocol involves all of the usual problems of software design, with the addition of interactions with other computers, lost information, delays, etc. An example of the design process for a simple control system application layer protocol (used in an educational context) is given in [6].

3 Control Network Configurations

3.1 Control loops and networking

Control systems may be represented in a layered manner. At the bottom is the controlled process. Next comes a first control layer in which each control loop acts on a single variable of the process. This is what we may call the "axis" layer in manufacturing. On top of this layer comes a second layer, the "axes" layer, which coordinates the actions on two or more control loops of the first layer. This coordination may use feedback from the first layer or may be independent of any feedback (feedforward). A third layer may then coordinate the actions of the coordinators of the second layer. Again, this may be done in a feedback or feedforward manner. This layering may be repeated to any number of layers even if in practice only a few layers are present. This cascading of feedback loops is sometimes referred to as multirate control [7]. Even if the presentation given above seems hierarchical, we do not assume any hierarchy. In practice, coordination between two or more entities at a given layer may

either be ensured by exchanging messages between the entities (decentralized case), or by using an entity of the next higher layer (hierarchical case). For the sake of the discussion, we shall assume a system with two layers as in Fig. 1. The first layer hosts all the single variable feedback controllers. The second layer hosts the set point generators. The conclusions that we will draw with this simplification may be easily generalized to a more complex case (we also assume periodic controllers).

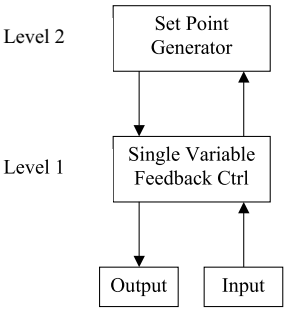


Fig. 1. Two level control system

If a control network is used in the system depicted on Fig. 1, it may carry the transfers between the sensors (or the actuators) and the loops at the first level. Alternately, it may also be inserted between the first and second levels and carry the information necessary to coordinate the level 1 control loops. In the first case (Fig. 2), the network will transport the values of the sensors to the loop controllers that will elaborate output values. These output values will be transferred on the network to the actuators. The network will also be used to synchronize the sampling on sensors that is required for temporal consistency reasons [8]. The main incentive for such a solution is the economy in cabling as well as the possibility of remote commissioning for the sensors and the actuators.

In the second solution (Fig. 3), the network is used to convey the synchronization information between the level 1 loop controllers as well as the necessary data. It will also carry the configuration and status data when necessary. This is the solution used when the loop controllers are decentralized close to the process or when the first solution is not available. The localization of the network in the system and the possible architectures of the elements at each level will create a number of different configurations and lead to varying constraints on the network.

3.2 Hierarchical systems

In this kind of system, two or more level 1 controllers are coordinated by a level 2 unit. This unit calculates the set point data for each of the controllers.

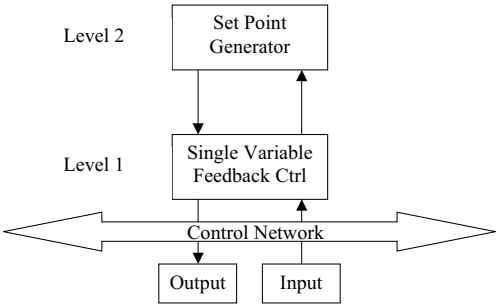


Fig. 2. Two level control system with control network within the control loop

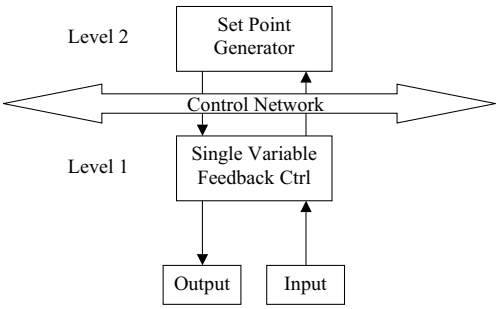


Fig. 3. Two level control system with control network above the control loop

Coordination may be ensured in two possible ways, explicit or implicit. In the explicit coordination, the level 2 unit periodically sends a message to all the controllers it manages. The message acts as a kind of heartbeat and instructs the level 1 controllers to start a new control period. The set point data may be sent with the synchronization message or separately. Implicit coordination is ensured through a common sense of time. The level 2 unit regularly sends the new set point data to each level 1 controller. The controller starts the new control period when its local clock indicates it. The local clocks are synchronized over the network using some clock synchronization algorithm [9].

3.3 Distributed control: peer-to-peer

The term "distributed control" is often used to describe hierarchical systems and refers to the fact that level 1 controllers are implemented as different hardware units. Our definition is different. In distributed control systems, level 1 units coordinate their operations without being synchronized by one or more units at level 2. An example is a distributed path controller for a machine-tool [11] or a robot. Each level 1 controller is given the description of the path and calculates the set point for the axis it controls. Coordination is

guaranteed by a common sense of time that may be obtained as in hierarchical systems. In some systems, in each coordination group, one of the level 1 units plays a special role by sending a kind of heartbeat message. The message is clearly a synchronization order but may also contain some index that is used by the other units to elaborate their own set point data. The index is often used as a pointer in a table of values that has been previously stored in each decentralized unit. This is often referred as "electronic cam" as it is similar to the master axis in a cam-based machine-tool.

3.4 Server-based systems

Server-based systems have the opposite structure of hierarchical systems. Instead of being commanded by the next hierarchical level, units fetch their orders from one or more servers.

3.5 Control-related network performance constraints

In the architectures described above, the network is either within the control loop (between the controller and the sensors and actuators) as in Fig. 2 or above the controller (Fig. 3). In the first case, the network transports the sensor input values and the actuator output values. In the second, it is used to provide the parameters and the set point data and to synchronize the loops. In case of cascaded loops (multirate control), both cases will be mixed and the constraints will add up. We shall here assume that all control loops operate periodically.

Two types of transfer incidents may occur, vacant sampling and message rejection. In the first case, a message does not arrive early enough for the application to use it. In the second, an application receives two input values when only one is expected. Both cases are related to variations in the transfer delay. Control networks should hence exhibit predictable transfer delays.

Network above the control loop

This case, depicted in Fig. 3, has sometimes been referred as feedforward ([12]). The network is used to synchronize the control loops and to transport the set point data.

Obviously, the set point data should arrive before they are used. The constraint is thus a bounded latency between the time the data is submitted to the network and the time it is delivered to the control loop.

Synchronization may be obtained in at least two ways: reception of the data or a special message and action based on synchronized clocks. In the first case, the transfer must take place at periodic instants because the control loops start their operations at reception of the message. This gives strict constraints on the periodicity of the transfers and the jitter (variation) in the period. As

a rule of thumb, the periodicity should be kept within a few percent of the required period. On the other side, if the latency must be constant, it may be long, as long as it is known, because the set point data may then be sent in advance. As guaranteeing periodic transfers with a small jitter on multiple messages is difficult, the set point data are seldom used to synchronize and synchronization is ensured by a single separate message. Note that sources of jitter are numerous, transfer errors, uncertainty in medium access control, variation in transfer duration of previous messages, etc., so message-based synchronization is seldom used in isolation.

The second option uses local clocks that are synchronized by adequate messages. Each control loop starts its period according to the value of its local clock. The local clocks are synchronized using one of the multiple distributed clock synchronization mechanisms ([9]). The advantage of this solution is that the strict message transfer jitter constraints disappear. This is paid by additional transfers of messages for the synchronization and some local computing to run the consensus algorithm. The set point data transfers must still be bounded in time. However, the messages used to synchronize the clocks need not be transferred periodically. There is an additional price to pay on the network. The accuracy of synchronization is directly linked to the uncertainty in the transfer delay over the network. The transfer delay is the time elapsed between the instant at which the first bit of the message leaves the source and that at which it arrives at the destination. This is different from the latency that includes the delays in the protocol layers. If the transfer delay is constant or if we are able to measure it, or calculate it, the accuracy of synchronization will be good. Otherwise, the accuracy will degrade. However, this constraint is less strict than the periodicity requirement of the first case.

In summary, control loops impose bounded latency and either periodicity with limited jitter or predictable transfer delays.

Network within the control loop

In this architecture (Fig. 2), the sensor input values are transported to the controller through the network. The outputs of the controller are later transferred by the network to the actuators. If there are multiple controllers, they may be synchronized through the same network as in Section 3.5. The same applies to the set point data. Here, we shall only deal with the input and output transfers.

The system operates in five steps. The sensors are sampled, the resulting information is transferred, the controller executes its algorithm, the results are transported on the network, and the outputs are reflected on the actuators. The usual assumptions are that sampling is periodic and the latency between sampling and actuating is constant and possibly small with regard to the sampling period. The controller need not execute strictly periodically, but it should run after receiving the input values and before the outputs values are transferred on the network.

Periodic sampling may be obtained using synchronized clocks or a special sampling message. The constraints are thus the same. However, when a controller uses several input values (position, speed, and acceleration, for instance), it expects that all were acquired at the same time. This means that all inputs should be sampled at the same time (temporal consistency). The sampling message must then be sent to all input nodes and received nearly at the same time (this might be a problem when switches are used).

The input and output data transfers are more or less constrained depending on the control solution. Let us depict two typical cases. In the first one, a single controller node executes N control loops for N axes. There is a single input value and a single output value for each axis. The objective is to limit the control latency below the duration of the sampling period. The inputs are sampled at the same time and result data are transferred one after the other on the network. As soon as a value is received, the controller executes the algorithm for this control loop. When this is done, the output value is ready to be transported on the network. With this solution, there is strict synchronicity between the network and the controller. The execution of a control loop may not start before the corresponding input value has been received. The output messages cannot be transferred before the controller has finished executing the corresponding algorithm. This translates into bounded latency for transfers. It may also create problems when the network is time triggered. If the controller desynchronizes and fails to produce an output value on time, the network will transport something invalid (possibly the value at the previous period). If the controller is time triggered, the same may apply at the inputs (vacant sampling).

Let us now assume that each control loop is executed on a separate node. Synchronization may be ensured as described previously. For input and output transfers, this case is the same as the first one. However, one may want to use smaller processors for the controller, and the execution time of the control algorithm will increase accordingly. Ideally, one would like to use each processor close to its capacity which corresponds to an execution time close to one sampling period. Then the constraints become stricter. Basically, all inputs must be transferred at the beginning of the period and all the outputs close to its end. The network has hence to withstand peaks of traffic and must be designed accordingly. There are other solutions to relax this constraint but they are beyond the scope of this chapter ([13]).

In summary, the constraints are the same as in the case for which the network is above the control loop. In addition, the instantaneous throughput may have to be much higher than the average one.

Common constraints

There are a few constraints that apply independent of the architecture:

- Notification of errors: Errors may occur in transfers. Many protocols trigger an immediate retry in this case. If the retry is not successful, additional

retries will take place until a maximum number is reached. Whether the protocol uses retries or not, the application should be notified of the failure.

- Temporal consistency: If required, the network should provide a way to know if different input values have been acquired at the same time.
- Absolute temporal consistency: This notion refers to the age of the information. The application should be able to determine if a given data was acquired recently or not. Usually, we need to know if the data corresponds to the last sampling instant or not.
- Event ordering: In some cases, the order in which events have occurred is important. This order may be obtained by time-stamping the event at occurrence time and relying on synchronized clocks to relate events that have occurred at different nodes. In some cases, it is sufficient to know at which period the event has occurred. In any case, the network should provide at least minimal support for this. If not, this should be reconstructed inside the application.
- Predictive behavior in case of overload: Unless the network is used well below its capacity, temporary overloads are likely to occur, e.g., in electromagnetic interference (EMI) perturbations. These are periods of time during which the network is given traffic exceeding its capacity. At such times, some networks delay low priority traffic. Some have a totally unpredictable behavior. For control application, the network should exhibit a predictable behavior.
- Amenability to analysis: Given the traffic requirements from the application, one expects to be able to find out if the network will comply with the given constraints before actually running the application. This means that the solution should be amenable to worst-case analysis.

4 Network Media

4.1 Media types

A number of transmission media have been used in networking: cables, waveguides, open space. From the transmission viewpoint, wired and wireless transmissions are the two main categories because they exhibit fundamental differences.

Wired transmission is usually based upon twisted pairs of copper conductors, coaxial cables, or optical fibers. Despite their high resistance to external electromagnetic fields, coaxial cables are today seldom used because of their cost. Because shielding is one of the conductors, ground loops may also appear when coaxial cables are used.

The most popular transmission media is the twisted pair. It is made of two metallic conductors that are twisted to mitigate the electromagnetic interferences. The pair may be shielded to provide an additional protection. A

cable may include one or more pairs that may be shielded as a whole or pair by pair. The most popular example of this cable is CAT 5 used in Ethernet.

When a higher degree of protection or galvanic isolation is necessary, optical fibers are a good solution. This comes with a price. Using optical fibers, multidrops must be excluded (in fact, they are possible using optical couplers but not economically viable). In other words, several network nodes may be hooked on the same twisted pair for instance when using the RS485 standard. With optical fibers, there must be a separate fiber from one station to the next.

Wireless transmission is an alternative when nodes are mobile. This may be the case of automatic guided vehicles in a workshop. Rotary joints on robots and machine-tools is another example. In this last example, the signals and the power are often transmitted using slip rings. There is a ring per conductor on the fixed part and a brush on the rotating part. This may be easily replaced by wireless transmission based on inductive coupling. One winding is on the still part (stator) and the other one on the rotating part (rotor). The rotation axis is the common axis of the windings. Longer ranges (longer than a few tens of centimeters) use either light, mostly infrared, or radio transmissions. Infrared is limited to line of sight (in reality this is a little better due to reflections on the walls), whereas radio waves may traverse walls and ceilings.

The main differences between wireless and wired transmissions are in their transmission range, bandwidth cost, security, and reliability. Kilometers may be easily achieved using cables. Wireless transmission is usually limited to meters for regulatory and space reuse reasons. For the same cost, wired transmissions offer effective bandwidths much higher than wireless bandwidths. While bit error rates around 10^{-7} to 10^{-9} are common on cables, rates up to 10^{-3} or 10^{-2} are not rare in wireless transmission. It may even be impossible to communicate in some cases. The main reason is that a cable may be considered as a private communication channel between two or more nodes. On the other side, open space is available to all. A perturbing device may totally hinder communication. Another common source of problems is fading. This happens when the waves that follow different paths interfere destructively at the receiver. This effect is wavelength dependent; that is why many wireless solutions either use a wide band (i.e., 802.11) or frequent carrier frequency changes (i.e., Bluetooth). Because a wireless signal cannot be constrained, security issues are more difficult as well. Eavesdropping or insertion of unwanted signals is much easier than in wired networks because a wireless signal may well propagate beyond its intended confines.

Despite all of the disadvantages enumerated above, wireless networking is wildly popular in the general computing environment because of the flexibility and cost and convenience savings generated from the hugely reduced need for fixed cables (there is still a need for some cables to connect the access points). Control system designers have been much more conservative in adopting wireless technologies except in special circumstances such as rotating parts of a machine. In particular the network speed variations and error

rates are a more serious problem in control systems than they are in general computing networks. However, as wireless technology matures, the lure of cost savings in cabling is affecting control systems as well. An interesting area of research that may have an important effect on control systems is that of very low powered, self-configuring networks [20]. Because of their low bandwidth, their impact would be in control systems with relatively long characteristic times such as building energy control.

4.2 Media requirements

Section 3 has pointed out the application requirements of control systems. The transmission medium plays an important role in complying with these requirements. At the physical layer, this translates into transmission error requirements. Each transmission error, unless there is an immediate recovery, will introduce some delay in transmitting the required information. The lower the error probability of the medium, the better the medium from that point of view. Here the cable itself, or the open space of a wireless system, is not solely responsible for errors. The transceiver properties (voltage levels, noise margin, impedance, etc.) also play an important role. A common requirement is that devices should withstand discharges of 1500 volts without being destroyed. It would be desirable that, under the same circumstances, communication could still take place. In summary, the combination of the transmission medium and the transceiver should be properly selected in order to keep the bit error rate sufficiently low.

There are obviously additional requirements such as the following.

- Topology: The cabled solution should be consistent with system design goals. For instance, Ethernet forces a tree topology in which each node is linked via a cable to a central point (switch or hub). In many cases, users would prefer a line topology in which the same cable runs from one node to the next and from the next to the following, etc.
- Connectors: They are in some cases the most expensive part of the solution. Connectors should be inexpensive but also robust to pulling and vibration. This only applies to wired solutions.
- Intrinsic safety: Some control networks are installed in environments that are explosive or inflammable. Special care must be taken in these circumstances.

5 Network Protocols

5.1 Interoperability

Two different nodes built by two different companies should be able to coexist on the same network and also to exchange information in a meaningful way

while complying with the requirements. This implies that they should use the same protocols at the different layers. Coexistence (without intercommunication) is possible, but only when the protocols at the lower layers are identical. Interoperability is sometimes guaranteed by tests, often specified by standards organizations, and realized at independent bodies. The guarantee may also be left to the manufacturers, who will test their devices against others. In any case, interoperability is an issue that needs serious consideration when selecting a solution. In the case of control networks, the issue becomes even more complex because the temporal constraints have to be assessed. In many cases, it is desirable or even necessary to include components from different manufacturers in the same control system network. Under these conditions, the importance of interoperability should never be underestimated.

5.2 Layering

It is the general acceptance of layered designs for network technologies that has been responsible for the high level of interoperability that is currently achievable. The idea of a layered representation of the protocols involved in a communication solution is not new and, as explained earlier in this chapter, a few different models have been presented. Here we will stick to the OSI seven-layer model for its better documentation. Layering is an important tool in understanding the functionality of a communication system. It may also be useful to build components that can be reused to create a solution. In each layer, various protocols may be used depending on the requirements of the application. For instance, at the application layer, one may use the Simple Mail Transfer Protocol (SMTP) to transfer mails or the Hypertext Transfer Protocol (HTTP) to access Internet pages. At the physical layer, a solution may be to use RS485 on shielded twisted pair as in Profibus or 2.4GHz direct sequence spread spectrum as in IEEE 802.15.4. Although, in principle, the choice of a solution at a given layer is independent of the option selected at another layer, this is not always the case. For instance, the IEEE 802 series of standards always describes the physical layer together with part of the data link layer (medium access control, MAC, adjudication of conflicts in access to the physical network medium). For control networks that have strict temporal requirements (see Section 3), there is even more interdependence. For instance, it is well known that the transmission control protocol (TCP) exhibits undesirable behavior using wireless media [21]. Finally, even if the solution is presented in a layered manner, the actual implementation may not follow the layering. In fact, in many cases, for efficiency reasons, layering is partially violated in the implementation. This violation of a design modularity principle is not uncommon in other areas of system design where, as in control systems, performance is extremely important. However, it should not be done lightly because portability, interoperability and maintainability are often sacrificed in order to achieve improved performance.

5.3 Connection between parts of a network

It is often necessary to interconnect parts of a complete network. The device that interconnects two or more subparts will be more or less sophisticated depending on the differences between the protocols used in the subparts.

- Repeaters are used when the protocols on all layers are identical on both sides. They are used to expand the distance covered by a link, either wireless or wired. They regenerate the signals received on one side and transmit them on the other side and vice versa. On some occasions, repeaters may also be used to interconnect a wireless cell to a wired link ([18]). Ethernet hubs are an example of repeaters.
- Bridges interconnect subnetworks using the same layer protocols above the data link layer. Both sides must also use compatible addressing information. For instance, IEEE 802.11 base stations interconnect an Ethernet-based link and a wireless cell. Similarly, an Ethernet switch is used to interconnect two or more Ethernet links.
- Routers operate at the network layer level. Their task is to find a route to convey a message from a source to a destination. Routers exchange information between themselves in order to find such a route. They can thus find an optimum path between two nodes, whereas bridges only use a subset of the available topology. The main difference between bridges and routers is that the latter are not transparent. Routers modify the packets they forward—in particular their address fields.
- Gateways are used when the protocols at the application layer are different on both sides. They translate the messages from one protocol to the other. For instance, connecting a Profibus or a CANopen network to the Internet, HTTP over TCP/IP, requires a gateway because the protocols are different at all layers.

5.4 Data link layer

The data link layer includes two subparts, the medium access control (MAC) and the logical link control sublayers. The MAC protocols define:

- How the medium is shared, as explained in Section 2.4;
- How the messages are organized in frames so that the beginning and the end of the messages can be recognized at reception;
- How the integrity of the messages is checked.

To check message integrity, additional information is appended to the information provided by the higher layers. The principle is that, at reception, alterations to the message can be detected with a high probability. Cyclic redundancy checks (CRCs) are examples of additional information. For instance, the CCITT-16 CRC is able to detect 100% of odd numbers of errors, double-bit errors, as well as any contiguous burst of errors shorter than 17 bits. Above 16 erroneous bits, more than 99.998% of the errors are detected.

Some error detection codes can also correct a number of errors. However, most of the time, the errors are corrected by asking for the retransmission of the information. This is usually done at the logical link control sublayer. This layer provides reliable message exchange between stations in a single link. It may also implement multicast and broadcast transmission. In the first case, the message is transmitted to a group of identified stations. In the second, it is sent to all stations.

Cyclic or even periodic transfers, when available, are implemented at the level of the data link layer. This is the case of Profibus, Multifunction Vehicle Bus (MVB), or factory instrumentation protocol (FIP). These standards share in common the use of buffers to control this kind of transfer. In a conventional transfer, a message is used to request some information from the provider of the information. The request goes up all the layers until it reaches the application. The application prepares a response that is sent back to the requester. The problem with this approach is that either the network is blocked until the response is given back, or the response must be transported in a separate transaction. To overcome this limitation, some standards disconnect the application behavior from the information transfer. When an application is ready to send information, it invokes the corresponding application layer service and the information is stored in a buffer at the data link layer (it is not transferred immediately). When the time comes to transfer the information, the content of the buffer is conveyed over the network to its destination. At the destination, it is again stored in a buffer at the data link layer. The application at the destination node may retrieve the information at any time by invoking the appropriate application layer service. The protocols at the destination node will then read the data link layer buffer and return the information to the application. There is no transfer involved at that time. The use of buffers allows better predictability in the transfers.

5.5 Higher layers

The next layer is the network layer. It is only required in large networks. The most popular protocol at this level is the Internet protocol (IP). Note that, contrary to some belief, interconnection with the Internet can be done without the use of IP. A gateway may be used instead.

The transport layer offers guaranteed transfers on a large network. It is only necessary when such a guarantee is needed and a network protocol is present. TCP is the most well-known transport protocol. However, it is sometimes not suitable for real-time transfers [15].

The session layer is used when recovery points must be inserted in the connection. It is never used in control networks.

The presentation layer is in charge of the representation of the information. It defines the transfer syntax. Conversion from the local syntax to the transfer one, and vice versa, is done in the presentation layer. For instance, some computers use the little-endian representation for the numbers while

some use the big-endian representation ("endian" refers to the order in which the individual bytes that make up an integer are transmitted). The presentation layer is in charge of the necessary conversion when these computers communicate.

In many cases, the presentation layer is included in the application layer. This is the only layer visible from the application. It defines the abstract view of the other nodes as seen from the local application. It also defines how the local application may operate on this abstract view. For instance, the file transfer protocol (FTP) defines the concept of files and directories. The available services allow for transferring files and for browsing directories. In the case of control networks, many of the application layers are inspired from the Manufacturing Message Specification (MMS) which defines a virtual manufacturing device that contains objects. Among the most useful objects are variables with services to read and write them, and programs with services to download and execute them.

The application layer plays an important role in the performances of the network. Many solutions are based on the client-server interaction. In this model, a node, the client, makes a request on another node, the server. For instance, the client may request the value of a given variable from the server. This often translates into a message carrying the request going from the client node to the server node. The response from the server is another message sent back to the client. As the server application is involved in the answer, it is very difficult to provide temporal guarantees. The producer-consumer model [19] is an alternative model. In this model, the application on the producer node delivers the value of the variable to the network. The value is stored in the data link layer and later transferred to all the subscriber nodes. When the application on a subscriber node wants to access the value, it is retrieved from the data link layer on this node without any transfer. It is thus much easier to offer real-time guarantees.

6 Network Reliability and Security

6.1 Reliability and security considerations

Reliability refers to the probability that a message on a network will arrive within the time constraints specified for the network. Security refers to the possibility that someone will purposely interfere with the network operation, to hinder delivery of information, to change the nature of the information, or to eavesdrop on the interchange.

In a broad sense, reliability subsumes security. However, in a practical sense, reliability usually refers to message delivery in the absence of purposeful disruption.

6.2 Physical integrity

The first line of defense for both reliability and security is the physical environment. The physical environment effects on message delivery can be categorized into two time scales: moment-to-moment traffic delays and the relatively rare catastrophic failures. Moment-to-moment issues usually revolve around electrical noise in the environment. Noise causes data corruption, leading to excessive retries, lost packets, etc., all of which can push message delivery outside the specified performance limits. In a control system, this can result in system failure.

Protection against noise requires proper understanding during the design phase of what the characteristics of environmental noise will be, and then the design of adequate shielding, effective grounding, and filtering so that network information loss is kept within specifications. Too often there is no realistic understanding of the noise levels that a network will actually see in an industrial environment. Optical data transmission is often specified in noisy environments because of its very high immunity from electrical noise.

Catastrophic failures in the information transfer part of a control system come from broken, disconnected, or shorted wires (other than shorts, optical systems are susceptible to the same failures). In discussing these issues for networked control systems, note that a major reason networks are used in control systems is to reduce the amount of discrete wiring. Thus, a networked control system, by virtue of having so much less wiring, is already far less likely to suffer such damage. With that said, standard design considerations such as strain relief for wires, connectors properly specified for the expected vibrational and environmental conditions, etc., are the primary routes to highly reliable systems.

Physical integrity can also be enhanced through network redundancy. Any number of independent communication paths can be provided so that a catastrophic failure in one will not disturb system operation. Because network information is packetized (see Section 6.3), sorting out duplicate information and detecting when one path has been damaged are quite straightforward. A major design issue in redundant systems, however, is ensuring that there is not a common point of failure, as for example, when a pair of redundant network cables is located close to each other so that both can be destroyed by the same mishap.

Wireless systems drastically reduce the number of wires and connections, but are much more susceptible to electrically or magnetically induced noise.

Protection of the physical perimeter is also the first line of defense for network security. Many control system networks can be completely local so that physical security can focus on people who are authorized to be in that space. Wireless networks, however, cannot be confined (except in the most extreme circumstances), so additional measures must be taken even if the network is technically local.

6.3 Abstraction

Because information is abstracted away from the physical layer in networks both reliability and security can be made arbitrarily strong—at a price.

Network reliability above the physical layer depends mainly on error detection and correction theory ([22]). Section 5.4 refers to some results of the application of error detection and correction within the data link layer of the protocol stack. In brief, extra information is added to each packet in such a way that errors can be detected, corrected, or both. The common parity check is the simplest of these schemes. With a parity check, one extra bit is added to whichever unit of information is to be checked (byte, word, record, etc.). The extra bit is added to make the total number of logic-one bits in the information unit specifically odd or even. When that information is received, the total number of logic-one bits is counted. If it is not odd (or even), then an error has occurred. This error detection scheme will find all one-bit errors and half of all multibit errors. It cannot correct any of the errors it detects. However, because the receiving node now knows that the packet is incorrect, it can request retransmission to get the correct information. Thus, the combination of a parity check with a retransmission facility will both detect and correct all one-bit errors.

The price to be paid for the added reliability is in network bandwidth and processing power. Assuming that the network is designed with proper bandwidth before any error detection or correction is added, an error detection and correction scheme will add to the size of the packet without adding any new information. Processing the packet will also require more computing power for the error detection and correction algorithms. Thus, the bandwidth will have to be increased to get the same information flow rate, resulting in a more expensive network as well as a more expensive processor.

Security uses the same structure—information abstraction and layering. Encryption is the tool of choice to ensure the security of information flowing on the network ([23]). Again, the stronger the encryption, the more the cost in network bandwidth and processing power.

Finally, both security and reliability also depend heavily on, as the saying goes in the world of automobiles, "the nut behind the wheel." Security, in particular, is peculiarly susceptible to individuals who are cleared to work on the system but, for some reason, wish to compromise it. However, even reliability can be seriously compromised by improperly performed maintenance, badly designed operator interfaces, etc.

6.4 Reliability and security: overview

The good news in designing reliable, secure networked control systems is that the very structure of networks lends itself to measured application of security and reliability solutions with predictable cost. The bad news is that the vulnerabilities are so broad, from a rodent chewing through a network cable to

an individual falsifying network information, that, unless a thoroughly integrated management and technical approach is taken, it is easy to leave serious holes.

7 A Survey of Network Solutions

7.1 Control networks

The first generation of control networks was designed in the late 1980s. Examples are FIP (CENELEC EN50170:3), SERCOS (CENELEC EN 61491), Interbus (CENELEC EN 50254), Profibus (CENELEC EN50170:2), LON (EIA-709.1) and CAN (ISO 11898). CAN has been successful in the automotive market except for critical functions. SDS (IEC 62026-5) and DeviceNet (IEC 62026-3) are variants of CAN designed for industrial automation. SERCOS has been designed for axis control and has its market in numerical controls. Interbus and Profibus-DP are commonly used for remote inputs and outputs in the programmable logic controller (PLC) market. LON is mainly used in building automation. WorldFIP (part of European Fieldbus standard EN50170) is now mainly used in safety related applications.

Most of these solutions have now reached their limits in terms of performance. They must be upgraded. The limited market size and the cost of new silicon development are major impediments for these new generations. On the other side, Ethernet, which was rejected in the 1980s for its lack of determinism, has been regularly improved toward higher performances and lower costs. For these reasons, the beginning of the century has seen the revival of Ethernet-based solutions. Solutions based on standard Ethernet use bandwidth limiting and subnet isolation to achieve satisfactory control performance. A number of different (and largely incompatible) solutions are currently proposed for control system use by different groups and industrial companies.

The advent of switched Ethernet with quality of service (802.1D and 802.1Q), better clock synchronization algorithms (IEEE 1588), and the use of traffic shaping pave the way to a standard solution based on Ethernet. The power-over-Ethernet standard (802.3af, [24]) further enhances industrial Ethernet by including power on the same cable as the network signal, further reducing the cable count.

An emerging network technology at the other end of the bandwidth spectrum is based on extremely low-power wireless connections. These networks are intended for applications requiring very low bandwidth but large numbers of very inexpensive nodes. An idea of the sizes contemplated is shown in the names used for this technology: “motes,” “dust,” “specks,” and TinyOS (an operating system that supports the self-configuration needed for such networks to be practical). While initially conceived for sensing applications, there are some research applications in control problems where network speed is not a big issue, such as building energy control ([16], [17]).

7.2 Other issues

Besides the aspects described up to now, networking solutions may offer various degrees of sophistication to ease installation, commissioning, and maintenance.

- Self-configuration is sometimes provided. Protocols such as UPnP (Universal Plug and Play), SDP (Service Discovery Protocol) or Jini are examples of efforts to provide various degrees of automatic configuration.
- Protocol analyzers are used to display the actual behavior of the networks.
- Design and simulation tools are used to give the expected performances of the solution.

8 Conclusions

The value of networking in control systems is indisputable at this point. However, if the design and integration of the networking are not done well, the performance of the control system can become completely unacceptable.

Design of a networked control system starts with functional and physical specifications. The functional specifications are of particular importance because they must be done in a control system context rather than a conventional networking context. Conventional network architectures specify average rate of data transfer. Even in video or audio streaming, the closest that common networks come to real-time applications, buffering is heavily used to provide a skip-free performance to the user. Control systems, by contrast, must specify delivery times. Getting this information may require considerable research of the performance literature or testing of prototype installations.

As we have noted, the choice of network technologies for control systems is quite large. Even when technologies that cannot meet the functional specification are eliminated, the choice is usually still large. Here the physical environment becomes a major factor. Control system environments are typically far harsher than standard business network environments.

Thus, reliability of the network becomes the next important part of the specification. Here the difference between analog delivery of control system information and networks becomes quite stark. Analog systems have a statistically based signal-to-noise ratio, but the information is always delivered (as long as there are no catastrophic failures). Because of the packet-based nature of networks, there is always some probability that a packet will not be delivered (however small). A fairly sophisticated stochastic-statistical analysis is required to relate these properties to control system performance.

Once reliability factors are assessed, adequate reliability can always be ensured if the error correction system is sufficiently robust. This reduces network useful bandwidth, however, so the design process then becomes iterative with the basic timing specification.

Because control system networks are often isolated, security issues can be less severe than in the much more open business network environment. Security issues do exist, however, and must be dealt with.

Finally, of course, are issues of cost. These are too complex to deal with in any detail here, but include physical cost (i.e., bill of materials costs), installation cost (keeping in mind that networked solutions usually provide substantial wiring cost savings over analog information transmission), cost of added processing power to deal with network protocols, and cost for problem-specific application layer software.

With proper attention to the network design and specification, and proper partitioning of the control system functionality to meet the full range of the system's timing requirements, networking expands on control system capability immeasurably.

References

1. Bijendra N. Jain, and Ashok K. Agrawala (1990), *Open Systems Interconnection: Its Architecture and Protocols*, Elsevier, Amsterdam, Netherlands, ISBN: 0444884904.
2. P. Bonner (1996), *Network Programming with Windows Sockets*, Prentice-Hall, Upper Saddle River, NJ.
3. K. Jamsa and K. Cope (1995), *Internet Programming*, Jamsa Press, Las Vegas, NV.
4. W. Stevens (1993), *TCP illustrated*, volume 1, Addison-Wesley, Reading, MA, ISBN: 0201633469.
5. CSMA/CD (Carrier Sense Multiple Access/Collision Detection) (2001), http://www.linktionary.com/c/csma.html
6. D.M. Auslander, J.R. Ridgely, and J.D. Ringgenberg (2002), *Control Software for Mechanical Systems*, Prentice-Hall, Upper Saddle River, NJ.
7. M. Törngren (1998), Fundamentals of implementing real-time control applications in distributed control systems, *Real-Time Systems*, vol. 14, pp. 219–250.
8. H. Kopetz (1991), Event-triggered versus time-triggered real-time systems, in *Operating Systems of the 90s and Beyond*, A. Karshmer and J. Nehmer (Eds.), Lecture Notes in Computer Science 563, Springer.
9. H. Kopetz and W. Ochsenreiter (1987), Clock synchronization in distributed real-time systems, *IEEE Computer*, vol. 36, no. 8, pp. 933–940.
10. W. Findeisen, A. Wozniak, P. Tatjewski, K. Malinowski, and M. Brdys. (1980) *Control and Coordination in Hierarchical Systems*, John Wiley & Sons, New York.
11. J.-D. Decotignie (1991), Distributed path and speed control in machine-tool axis motion, *Proceedings IECON '91*, 28 Oct.–Nov. 1991, vol. 1, pp. 772–777.
12. J.-D. Decotignie, D.M. Auslander, and M. Moreaux (1996), Fieldbus based integrated communication and control systems-Architectural implications, *Advanced Motion Control*, 1996. AMC '96-MIE. Proceedings., 1996 4th International Workshop on, Volume 2 , 18–21 March 1996, pp. 541–546.

13. S. Koppenhoefer, J.-D. Decotignie, and D.M. Auslander (1996), Fieldbus based integrated communication and control systems, Symposium on Control, Optimization and Supervision. CESA '96 IMACS Multiconference. Computational Engineering in Systems 1996, pp. 1316–21, vol. 2.
14. M. Farsi, K. Ratcliff, and M. Barbosa (1999), An overview of controller area network, *Computing and Control Engineering Journal*, June 1999, pp. 113–120.
15. S. Iren, P. D. Amer, P. T. Conrad, The transport layer: tutorial and survey, *ACM Computing Surveys*, vol. 31, pp. 360–404, Dec. 1999.
16. Web site with extensive references: http://www.tinyos.net/media.html.
17. Web site of the SpeckNet consortium: http://www.specknet.org/.
18. Ph. Morel (1996), Integration d'une liaison radio dans un rseau industriel, Ph.D. Thesis 1571, Swiss Federal Institute of Technology (EPFL), Lausanne, 1996.
19. J.-P. Thomesse (1993), Time and industrial local area networks, *Proc. of 7th Annual European Computer Conference on Computer Design, Manufacturing and Production (COMPEURO'93)*, pp. 365–374, Paris-Evry (France).
20. Guest Editors' Introduction: Overview of Sensor Networks D. Culler, D. Estrin, and M. Srivastava (2004), *Computer*, Vol. 37, No. 8, pp. 41–49
21. F. Anjum, and L. Tassiulas, Comparative study of various TCP versions over a wireless link with correlated losses, *IEEE/ACM Transactions on Networking (TON)*, Vol. 11, Issue 3, June 2003.
22. R. Blahut, *Theory and Practice of Error Control Codes*, Addison-Wesley, Reading, MA, 1983.
23. *Handbook of Applied Cryptography*, by Menezes, van Oorschot, and Vanstone, CRC Press, Boca Raton, FL.
24. IEEE Std. 802.3af-IEEE Standard for Information Technology-Telecommunications and information exchange between systems-Local and metropolitan area networks-Specific requirements, 2003. Also, http://www.ieee802.org/3/af/

Part II

Hardware

Basics of Data Acquisition and Control

M. Chidambaram

Department of Chemical Engineering
Indian Institute of Technology, Madras
Chennai 600036, India
chidam@iitm.ac.in

1 Introduction

Data acquisition was traditionally carried out manually by noting down the readings of various instruments at specified times. Traditional instruments required extensive time and specific skills for adjusting the measuring range and for saving and documenting the results. Later, electronic recorders recorded the data acquired on paper plots. Computer-based data acquisition replaced paper records by digital data acquisition and storage. The automatic collection of data from sensors, instruments and devices in a factory, laboratory or in the field is called data acquisition. The use of computers in various aspects of data collection, control and analysis over the past few years has revolutionized modern-day research, development and manufacturing.

Real-world signals are not compatible with the binary formats used by the microcomputer. Hence, there is a need for signal conversion before and after processing. In this chapter, the additional circuits required for connecting real-world sensors to the computer and for connecting the computer to the final control elements are presented. Real-time interfacing is a general term used to describe the aspects of connecting a computer with a real-world process for communicating data between the two. A data acquisition system is a collection of add-on hardware and software components that allow the computer to receive real-world information from sensors. The data will be stored for plotting, processing and writing to a file. The data acquisition system may be considered as a monitoring system. A data acquisition and control (DA&C) system sends data to the real-world system (mainly to actuators such as solenoid valves, pneumatic valves, relays, motors, etc.). Fig. 1 shows the block diagram representation of a data acquisition system. The flow of information in a typical DA&C system can be described as follows.

1. The input transducers measure some property of the process.
2. The output from the transducers is conditioned (amplification, filtration, etc.).

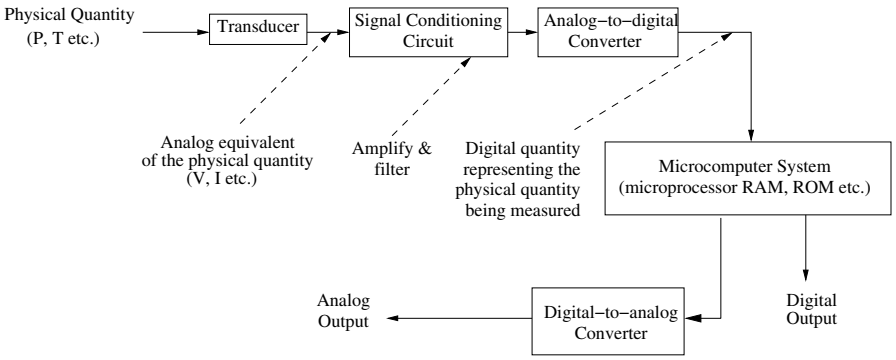


Fig. 1. Data acquisition system

3. The conditioned analog signal is digitized using an analog-to-digital converter (ADC).
4. The digital information is acquired, processed and recorded by the computer.
5. The computer may then calculate the control signals to the process.
6. The digital control signals are converted to analog signals using a digital-to-analog converter (DAC).
7. The analog signals are conditioned appropriately for a final control element.
8. The final control element interacts with the system by changing the value of the manipulated variable.

2 Signals

The majority of signals are analog signals. Analog signals are defined over continuous time intervals and assume a continuous range of amplitude values. An ADC module changes an analog input signal into a binary output code. For an input analog voltage or current, there is a corresponding proportional binary output. If an 8-bit ADC has a 0 to 2.56 V input signal range, then 0 V input could produce an output word of 00000000, while the +2.56 V level seen at the input would produce an output of 11111111. Sampling is necessary in real-time digital processing to (i) allow for the processing time in ADC and control law calculations and (ii) share expensive computers or other equipment among many signals.

The DAC receives the binary value (digital signal) from the computer (CPU) and converts it into an analog voltage (or current) that can be used for actuating an external final control element or device or for displaying the digital data from the microcomputer to the equivalent analog data. DACs and ADCs are available for the IBM PC in a number of forms. The most common form is as one of the components on a general-purpose laboratory

data acquisition interface card, which plugs into one of the slots of the mother board of the PC. This card (refer to Fig. 2) typically contains an ADC, one or more DACs, and sections for digital input/output and counting/timing. ADC specifications are given by conversion time, resolution and accuracy.

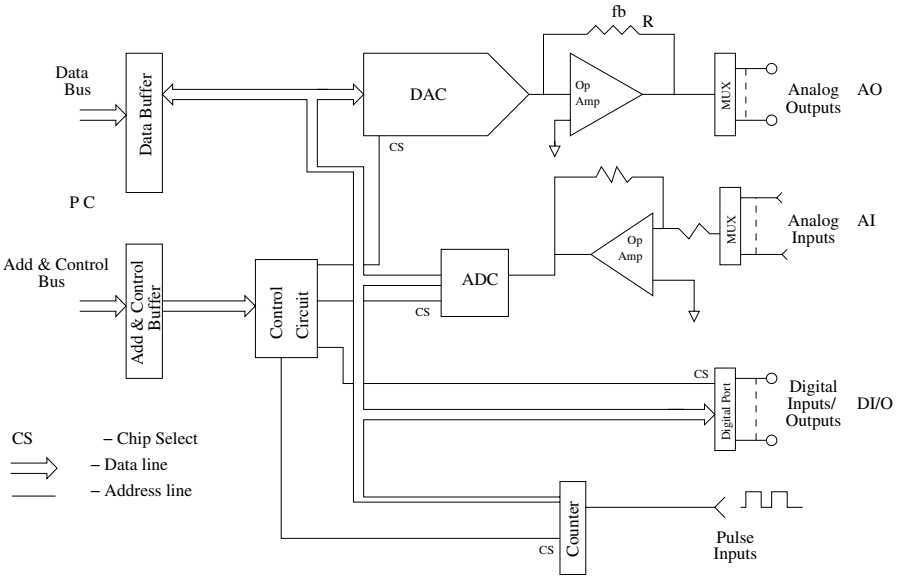


Fig. 2. PC add-on card for DA&C

Conversion time is the time required to complete a conversion of the input signal. It establishes the upper signal frequency limit that can be sampled without aliasing:

$$f_{max} = 1/(2*\text{conversion time}).$$

Resolution is the number of bits in the converter. The resolution is the smallest analog signal for which the converter will produce a digital code. It may be given in terms of the full scale input signal,

$$\text{resolution} = (\text{full scale signal})/2^n,$$

where n is the least significant bit of the ADC. Resolution relates the smallest signal to the full scale value. Accuracy relates the smallest signal to the measure signal. Accuracy is given by percentage and describes how close the measurement is to the actual value. When converting analog information into digital information, the resolution of the data conversion is an important consideration. ADCs have 8-bit, 10-bit, 12-bit or 16-bit precision, For example, an 8-bit converter within a range of -5 V to $+5$ V will slice the total voltage span (10 volts) into 255 steps. Each step or resolution would equal 39.2 mV.

A 12-bit converter will slice the 10 volts span into 4095 slices. The step size or resolution is 2.4 mV. The higher the resolution, the higher is the cost of the ADC, however.

3 Interfacing Input Signals

3.1 Analog signal conditioning

Table 1 provides a list of transducers that give an analog signal as the output. The analog signal from a transducer must be conditioned by a signal conditioning circuit to meet the input requirements of an ADC input. Circuits such as operational amplifiers, bridges and comparators are used for analog signal conditioning. Signal conditioning involves the following steps: (i) signal amplification, (ii) isolation,(iii) multiplexing, (iv) noise filtering, (v) transducer excitation, (vi) use of simultaneous sample and hold and (vii) anti-aliasing filtering.

Thermocouple
Resistance temperature detector (RTD)
Thermistor
Strain gauge transducers (for pressure, load force, torque measurements)
Potentiometer/resistance output 0–20 mA or 4–20 mA current signals
Millivolts/volts output from pH electrode, humidity sensors, level sensors, flow sensors, pressure sensors, conductivity electrode, chromatography

Table 1. Sensors giving output as analog signal form

3.2 Signal conditioning circuits

Operational amplifiers (op-amps) are used to amplify a signal or difference between two signals. The input from many instruments is at low signal levels. An amplifier is thus often required. In order to take full advantage of the resolution of the ADC, the amplifier must be designed to provide outputs over the entire range of the ADC. For example, a signal from instruments that vary over the range of 0 to 10 mV should be amplified by a factor of 1000 to provide 0 to 10 V output (a typical range required for an ADC).

There are two basic methods for carrying out the amplification steps: using an external amplifier or buffering an ADC with a built-in amplifier. The external amplifier can be placed inside the instrument or at least very close to it, thereby reducing the effect of noise. External amplifiers are more expensive than the internal systems, because of added hardware (excluding the power supply unit). They are slower than the internal systems. Even though the data may be collected at a faster rate, transmission of the digitized information to the host is limited by the transmission speed. The use of an amplifier built into the ADC board has advantages: it is convenient and it avoids possible

grounding and impedance matching problems that are encountered with an external amplifier. Many of the commercially available ADC boards give the user the option of an on-board amplifier. In some cases, the gain of the amplifier is fixed; in others, the gain can be modified by changing the hardware on the ADC board (i.e., changing the size of the resistor). It is also possible to change the gain in the software between data conversions; this is referred to as programmable gain. However, it may slow down data collection because of the need to check the size of the input signal and to reset the gain when appropriate. Its chief advantage is the added flexibility it gives to the user in selecting the gain when the experiment is proceeding.

An ADC for PCs may use a 12-bit resolution in a fixed analog signal range of −10 V to +10 V. At best the ADC will be able to distinguish between analog voltages 5 mV apart (20 V/2^{12}). A transducer, for example, a thermocouple, produces a small voltage, perhaps in the range of 0 to 30 mV. As a result, the 12-bit ADC will produce only 6 out of a possible 4096 integers to describe the temperature being recorded by the thermocouple. If the thermocouple covers 0 to 2000^{o}C, then the ADC can only resolve 333^{o}C (2000/6) differences in temperature. This is not at all acceptable. An external analog amplifier should be used to amplify the thermocouple signal to provide more resolution. An amplifier with a gain of 333 will amplify the thermocouple signal so that the full temperature range (0 to 2000^{o}C) will correspond to 0 to 10 V at the ADC input. In this case, with the external amplifier the ADC can resolve 0.98^{o}C (2000/2048) difference in temperature.

3.3 Input signal buffering

Some interfacing applications require impedance matching between the sensor and the input terminal on the DA&C board. A unity gain operational amplifier is connected between the sensors and the DA&C board. The high input impedance of the operational amplifier minimizes the loading in the sensors and prevents signal degradation. The low output impedance of the operational amplifier is ideal for loads such as motors, recorders or connection to a DA&C board. External amplifiers can also be used as buffers to electrically isolate sensors and transducers from the PC data acquisition systems. This is done so that the noise or signal in the PC ADCs does not affect the transducer responses. Other sensors that provide large signals such as piezoelectric transducers may damage the PC if they are not buffered or electrically isolated from the PC. If the analog signal produced by transmission is too large for direct conversion to an ADC, then the signal must be reduced to a lower value. This scale down can be achieved by dropping the required voltage across a suitable resistor.

3.4 Offset elimination

An offset occurs with a variety of sensors and transducers. An offset voltage is an unwanted voltage that is produced by a sensor though the sensor

theoretically should be producing zero volt. Offset occurs due to inaccuracies in manufacturing and calibration of sensors. The offset can be eliminated by using a signal conditioning circuit such as an operational amplifier connected in a summing application.

3.5 Filters

A filter is required in signal conditioning circuits to eliminate unwanted frequencies (of the noisy corrupted signal). There are two types of filters: passive and active filters. Passive filters are designed using standard discrete components such as resistors, capacitors and inductors. Filters are classified as low pass filters, high pass filters and band pass filters depending on the range of frequencies over which the signal is useful. When we use derivative action in a proportional-integral-derivative (PID) controller, it is desirable to filter the noisy signal and remove the high frequency noise before the control action is calculated.

ADCs are often connected to analog signals that contain frequencies higher than the sampling rate of the ADC. Many analog signal transducers like photodiodes and piezoelectric sensors generate high frequency noise, often referred to as harmonic distortion. If these transducers are sampled using an ADC, the high frequency components in the signal will increase the amplitude of the low frequency components of the acquired data samples. This error is called aliasing error since the high frequency components of the signal appear erroneously in the sampled data as low frequency components. A general rule in the laboratory is to use A/D conversion frequencies that are at least 10 times higher than the highest frequency component in the signal. To avoid aliasing errors, many PC ADCs include an anti-aliasing filter with the ADC circuitry. These are usually low pass analog filters (refer to Fig. 3).

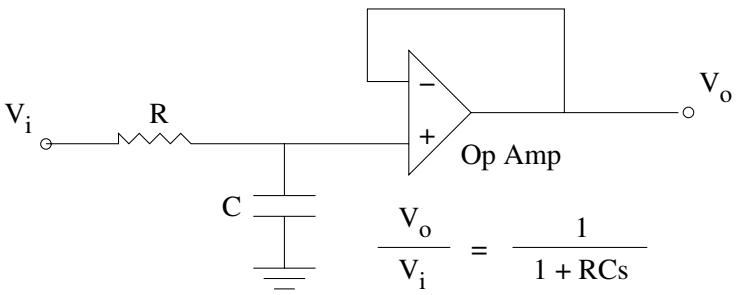


Fig. 3. First-order low pass filter

3.6 Bridge circuit

This circuit is used to measure the small resistance changes caused by a resistive transducer. Transducers using Wheatstone bridges include strain gauges, thermistors, resistance temperature detectors (RTDs) and almost all resistive transducers. A number of such primary elements convert changes in the measured variable into small changes in the resistance of the element. Strain gauge force transducers, strain gauge pressure transducers and resistance temperature detectors are three examples. The use of a bridge circuit is the traditional method of measuring small changes in the resistance of an element. The operation of a bridge is classified into two categories, balanced and unbalanced operation. In the balanced operation, the resistance of the sensor is determined from the values of three other resistors whose values are known with precision. In the unbalanced operation, the change in the sensor resistance from a base value produces a small difference between two voltages. A differential amplifier is used to amplify the difference between the two voltages (refer to Fig. 4).

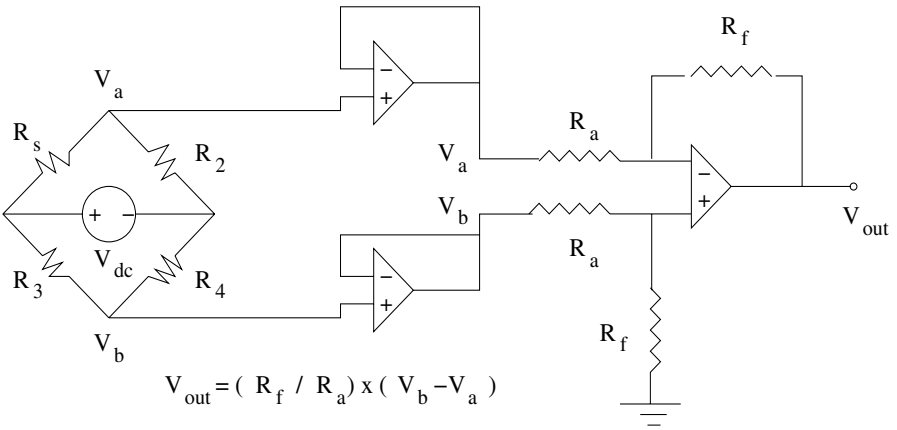


Fig. 4. Unbalanced Wheatstone bridge and instrumentation amplifier circuit

3.7 Isolation

In many situations, isolation of the control and controlled signal is desirable or even necessary. For example, the digital output from a computer may be used to switch the power device on and off. The instrument at the other end may be used to run a furnace or a pump or anything else. While it is quite possible to use the digital output to directly switch a thyristor or a triac, it is not desirable since a failure of the thyristor may in certain cases lead to feeding the main to the computer with disastrous results. This can be avoided

using a solid state (optically coupled) relay or a magnetically coupled relay or reed relay circuit. Another situation may be an electrically noisy system where it may become necessary to isolate the controller and the controlled system to avoid feedback of noise into the controllers.

3.8 Current loop

If analog sensors are located at a considerable distance from the computer or controller, then the analog signal is transmitted as a current signal rather than as a voltage signal. The standard associated with a current loop circuit is a 4 to 20 mA circuit. The minimum current of 4 mA always in the circuit can be used to check circuit integrity. A signal above 20 mA indicates that the system is malfunctioning. A special module transmitter does the conversion of the voltage signal to the 4 to 20 mA current signal. The current loop provides a high degree of noise immunity and avoids the loading difficulties caused by having more than one receiving device in the loop. For example, a current loop can be connected to a DA&C board as well as to a strip chart recorder, or other instrumentation devices.

Most of the actuators also work with a 4–20 mA control signal. The main advantage of the current loop operation is that the signal is not affected by a long cable length. Also, this leads to standardized operation, making it simpler to develop interfaces. Quite often the signal converter that converts the transducer output to a linear 4–20 mA signal induces electrical isolation of the input from output. Similarly, many analog elements such as valves are operated by a 4–20 mA signal through current-to-pressure (I/P) converters. For A/D conversion, a short resistor (normally 250 ohms) is required at the ADC end to convert the loop signal into a voltage. The 4–20 mA signal is converted into 0–5 volts, which is easy to handle with single-ended inputs. General circuit diagrams for voltage-to-current and current-to-voltage conversion are given in Fig. 5.

3.9 Sample and hold circuit

The sample and hold (S&H) circuit shown in Fig. 6 consists of an electronic switch, a capacitor and a buffer amplifier. The circuit is switched to the sample mode using a control line from the microprocessor port and this closes the switch, forcing the capacitor to charge to the value of the analog input. After a short interval the circuit is put into the hold mode and the switch is opened. The capacitor then retains the value of the analog signal to which it had been charged. During the hold mode, the ADC performs the conversion. Since a high quality capacitor and buffer amplifier are used in the S&H circuit, the voltage signal presented to the ADC will remain constant during the conversion time. As stated earlier, the S&H circuit is used at the input of the ADC where it keeps the analog signal constant during the data conversion process.

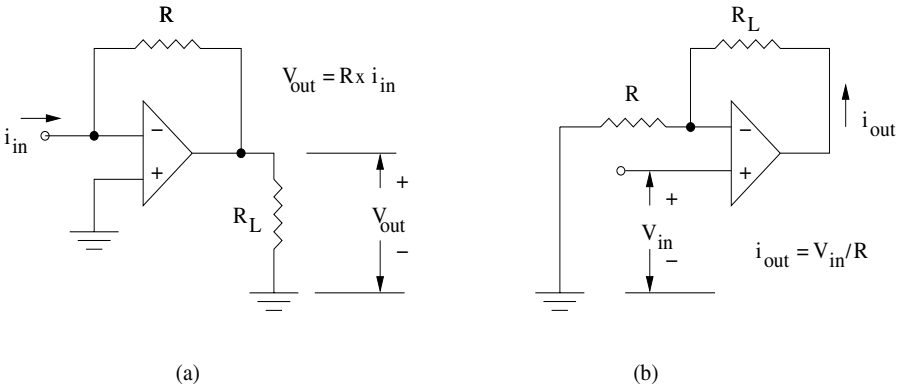


Fig. 5. (a) Current-to-voltage converter and (b) voltage-to-current converter

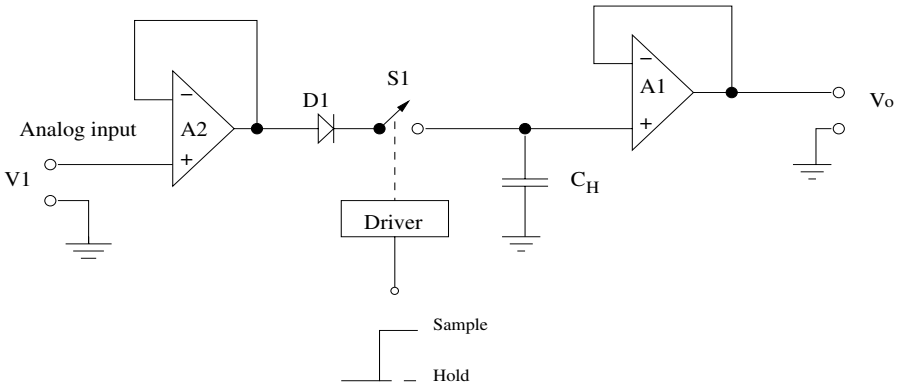


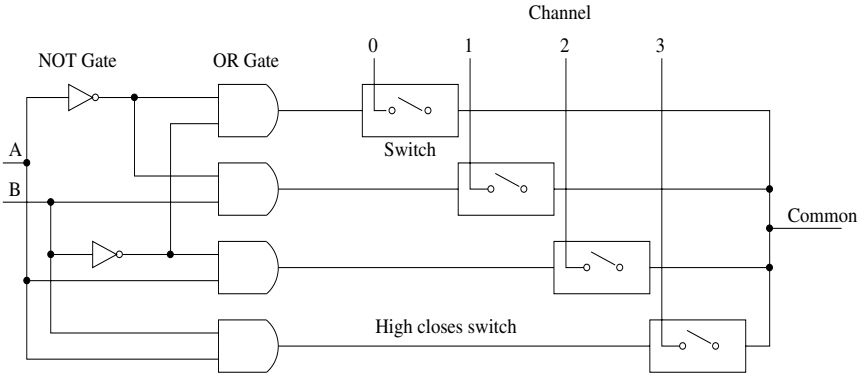
Fig. 6. Sample and hold circuit

3.10 Multiplexers

A multiplexer (MUX) is an electronic equivalent of a rotary switch. When in operation the common line is connected to one of the eight channels. The MUX can be disabled, in which case the common is not connected to any of the channels. A four channel MUX should only require two binary lines to select any of the four different channels. Fig. 7(a) shows a gate circuit that in conjunction with the four switches forms a rudimentary MUX. The channels selected by the four combinations of A and B are shown in Fig. 7(b).

3.11 Selection of a unipolar or a bipolar input range

For most signals, the input should be set to bipolar. In this mode, the DA&C board will accept both positive and negative voltages. Typical ranges are -5 to $+5$ V and -10 to $+10$ V. In cases where the signal is to be guaranteed



(a)

A	B	Channel
0	0	0
0	1	1
1	0	2
1	1	3

(b)

(a) A gate circuit for an elementary four–channel MUX

(b) Channels selected by various combinations of control lines A and B

Fig. 7. Multiplexer

never to be negative, then setting the input to unipolar mode will improve the resolution by a factor of up to 2. On the other hand, if the input is set to unipolar and the signal goes to negative, it will be severely distorted by the recording process. Changing between the unipolar and bipolar modes is carried out by setting a switch or a jumper on the data acquisition board itself or via the computer.

3.12 Selection of single-ended or differential inputs

In most cases, the inputs should be set to single ended. Differential inputs are better at rejecting noise, particularly if the signal is small or if it is connected to the data acquisition system by long cables. However, differential inputs are more complex to wire up, and only half the number of input channels will be available compared to that in single-ended mode. Signals or noises that are the same in both the inputs of a differential amplifier (relative to ground) are called common-mode signals. The differential amplifier will amplify the differential-mode signals and reject the common-mode signals. How well the amplifier does this is measured by the common-mode rejection ratio (CMRR).

More complex devices such as pH meters, ion-exchange probes and glass microelectrodes are frequently used but all come complete with their own special amplifier, and connecting the amplifier output to a data acquisition system is not difficult.

4 DA&C Add-on Card

Plug-in data acquisition and control (DA&C) boards (called add-on cards) are the fastest growing instrumentation option and represent the heart of instrumentation. Plug-in DA&C boards are available for many popular computers. There are also cards for a single purpose (i.e., dedicated to a particular function), for example, carrying out only A/D or only D/A conversion. The single-purpose card is low cost, simple to use and easy to program. If we need to use more than one function, then buying cards for each function will add to the cost; also, that many expansion slots may not be available in the PC. Another class of interface boards, known as multifunction cards, have various combinations of analog, digital and pulse inputs and outputs. Many cards have programmable channel sampling and conversion modes (separate gain for each channel). One card often can perform a variety of functions including A/D or D/A conversion, digital input-output (I/O) and counter/timer operations (refer to Fig. 2). The multifunctional card needs one expansion slot, and the cost of the card is cheaper compared to buying several single function cards. However, the multifunctional card requires several consecutive I/O addresses. Most of the multifunctional laboratory cards are built with 4 to 16 analog input channels; even 4 channels are certainly sufficient. Because the channels are multiplexed (there are 6 channels but one A/D), the sampling rate quoted for A/D is the maximum rate for all the channels combined. If some channels are not used, then the sampling rate is decreased accordingly.

The bus expansion slot allows a card to communicate electrical signals directly with the central processing unit (CPU) over the PC bus. Control of this process is achieved through software that addresses the card either as a memory location or as an I/O port. Memory address referencing is used for most process control I/O applications because of its speed and larger address space. The speed at which a card can communicate data with the CPU is a function of three items:

1. Clock frequency of the processor chip (i.e., 50 MHz, 66 MHz, 75 MHz)
2. Bit length of the processor (i.e., 16 bits, 32 bits)
3. Bit length of the bus (i.e., 8 bits, 16 bits, 32 bits)

The ranges of PC expansion boards currently available from a large number of manufacturers include:

1. Analog I/O card with up to 16 analog inputs (and up to 4 buffered analog outputs)
2. Digital I/O cards with direct transistor-transistor logic (TTL)-compatible inputs and outputs
3. Digital I/O cards with optically isolated inputs and outputs
4. Digital I/O cards with buffered I/O lines
5. Digital output cards with reed relays

6. Digital output cards fitted with solid state relays for AC or DC power control or triac for AC power control
7. IEEE-488/ general-purpose interface bus (GPIB) interface card
8. Multifunctional I/O cards (offering mixed analog and digital I/O facilities)
9. Bus expansion cards (which interface external card frames or mother board)
10. Thermocouple interface cards
11. Stepper motor controllers
12. Specialized instrument cards (e.g., digital multi-meters, counters/digital frequency meters)

It is easy to construct a PC-based process control system by selecting off-the-shelf modules. Only when dealing with specialized applications is it necessary to use one's own dedicated I/O cards and or external signal conditioning boards.

4.1 Digital I/O

Data acquisition systems often include the facility to deal directly with digital signals. Digital I/O circuits move information from the real world into the computer and from the computer into the outside world. The circuitry to make a digital input is similar to that needed to make a digital output. Thus, the two functions are combined into one; hence the term digital I/O. During the configuration of the add-on card, we have to set which channels are input channels and which are output channels. A variation of the basic digital I/O port is the counter and timer. Most data acquisition systems with a digital port have one or two counter/timers. These are digital inputs and work over the same range of voltages as standard digital inputs. Like basic digital I/O lines, the counter/timers can be configured either as inputs or outputs. In counter mode, the line is configured as an input. The counter counts pulses applied to it; a single pulse is counted when the digital input goes from low to high and back to low again (or from high to low to high). The number of pulses is then read by the host computer, which can also clear the counter (set the counter to 0). The counters in the I/O circuitry are digital and binary. They are specified by the number of bits in each counter. In timer mode, the counter/timer acts like a digital stop watch. The counter is connected internally to a circuit that generates a stream of pulses of known frequency. This circuit is often referred to as a clock.

4.2 Digital-to-analog converter

The heart of most DACs, is a current summing node where the currents are selected by digital inputs. Consider, for example, the 3-bit DAC (refer to Fig. 8). A_2 and A_0 correspond to the most significant bit (MSB) and least significant bit (LSB) of the digital input, respectively. The largest resistor

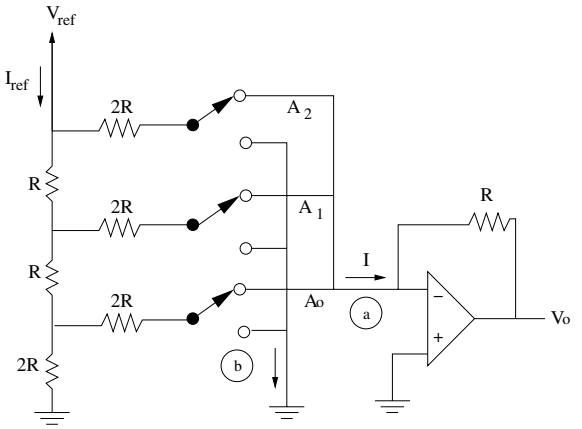


Fig. 8. An R-2R ladder DAC

value is only twice that of the smallest, and hence this network is called an R-2R ladder DAC. The right-hand side of all of the 2R resistors is at 0 volt and hence the total resistance of the network is R ohms. The reference current I_{ref} is V_{ref}/R . The current at the summing node I, becomes

$$\begin{aligned} I &= (V_{ref}/2R)(A_2{+}0.5A_1{+}0.25A_0) \\ &= V_{ref}(4A_2{+}2A_1{+}A_0)/(8R) \end{aligned}$$

Hence, the output voltage V_0 is given by

$$V_0 = -IR = -V_{ref}\ (4A_2{+}2A_1{+}A_0)/8$$

The output voltage range is from 0 to $-7V_{ref}/8$ volts in steps of $V_{ref}/8$. This step size corresponds to the LSB.

4.3 Analog-to-digital converter

The successive approximation ADC is very widely used because it is relatively fast and cheap. It uses a DAC in a feedback loop as shown in Fig. 9. When the start signal is applied, the S&H amplifier latches the analog input. The control unit then begins an iterative process, where the digital value is approximated, converted to an analog value with the DAC and compared to the analog input, the end signal is set by the control unit and the correct digital output is available at the output. If n is the resolution of the ADC, it takes n steps to complete the conversion. More specifically, the input is compared to combinations of binary fractions (1/2, 1/4, 1/8, ..., $1/2n$) of the full scale (FS) value of the ADC. The control unit first turns on the MSB of the register, leaving all lesser bits at 0, and the comparator tests the DAC output against the analog input (refer to Fig. 10). If the analog input exceeds the DAC output, the MSB is left on (high); otherwise, it is reset to 0. The procedure is then applied to the next lesser significant bit and the comparison is

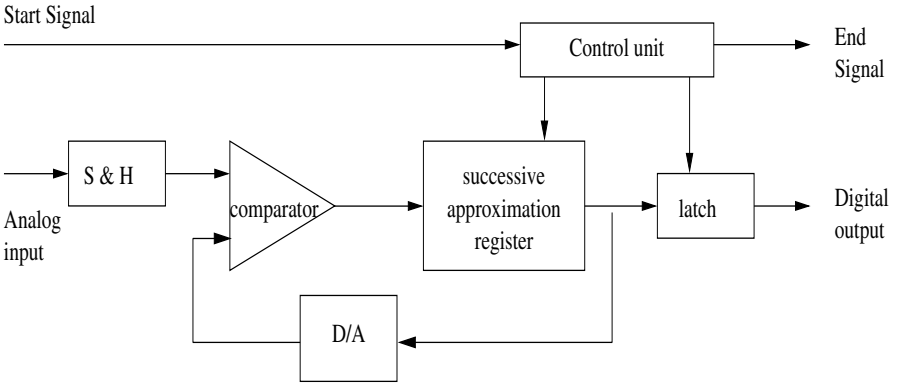


Fig. 9. Successive approximation ADC

made again. After n comparisons have completed, the converter is down to the LSB. The output of the DAC then represents the best digital approximation to the analog input. When the process terminates, the control unit sets the end signal, signifying the end of the conversion. Typical conversion times for 8-, 10-, and 12-bit successive approximation ADCs range from 1 to 100 s.

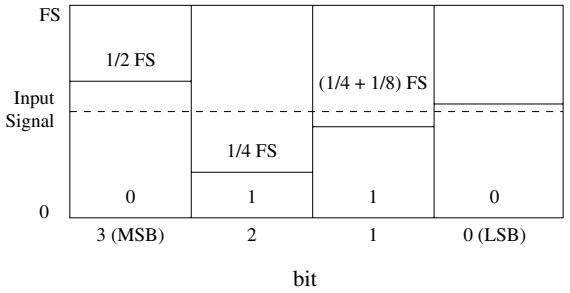


Fig. 10. 4-bit successive approximation A/D conversion

4.4 Buffered DA&C

There are standard techniques that take some of the load off the computer and transfer it to the ADC hardware. One of these techniques frees the computer from the task of fetching and storing each sample from the ADC when it is ready. Instead, the data are held in a temporary electronic store called a buffer. A buffer is a circuit containing some memory chips and it is placed in between the ADC and the computer. In some cases, it is physically located on the DA&C board—a buffered DA&C board. The ADC and buffer circuits collect data on their own without any intervention from the computer until the buffer

is full: the computer then copies all the numbers from the buffer into its own storage system in one go. Moving a block of data is much more efficient than moving samples one at a time, and while the buffer is filling, the computer continues with data analysis. Buffers are also useful when collecting large amounts of data rapidly because the buffer circuit can collect samples much faster than the computer. The second technique for handling large amounts of data at high sampling speeds is to use direct memory access (DMA). With DMA, the DA&C board stores the samples directly in the computer memory area (RAM) without any intervention from the computer CPU. This technique is often combined with buffering because it allows the DA&C card to use the host computer's RAM as a buffer rather than having to have memory chips on the A/D card itself. This lowers the cost of the DA&C card.

5 Configuring Options and Settings

DA&C adapter cards often have configurable options that must be set for the adapter card to function properly. Examples include: interrupt request (IRQ), base I/O port address and base memory address. Sometimes it is possible to configure DA&C adapter card settings in software, but these settings commonly must match jumper or dual in-line package (DIP) switch settings configured on the DA&C adapter card. DA&C card documentation should be referred to for DIP switch settings. Many newer DA&C adapter cards use Plug-and-Play (PnP) technology, which makes manually setting the adapter card options obsolete. The operating system configures the hardware device automatically.

5.1 Interrupt requests (IRQ)

Interrupt request lines are hardware lines over which devices such as input/output ports, the keyboards, disk drives, DA&C adapter and computer network adapter cards can send interrupts or requests for services to the computer's microprocessor. Interrupt request lines are built into the computer's internal hardware and are assigned different levels of priority so that the microprocessor can determine the relative importance of incoming service requests. Each device in the computer must use a different interrupt request line or interrupt (IRQ). The interrupt line is to be specified when the device is configured.

5.2 Base I/O port

The base I/O port specifies a channel through which information flows between the computer's hardware (such as DA&C adapter card) and its CPU. The port appears to the CPU as an address. Each hardware device in a system must have a different base I/O port number.

5.3 Base memory address

The base memory address identifies a location in a computer's memory (RAM). This location is used by the adapter card as a buffer area to store the incoming or outgoing data. Some adapter cards contain a setting data that will specify the amount of memory to be set aside for storing data. For example, for some cards we can specify either 16K or 32K memory. We have to specify some appropriate part of the main memory as the starting memory or base memory for the card. To set the devices or card for this base memory address we have to set appropriate DIP switches in the add-on card. The base memory address is to be specified as a hexadecimal number. The memory required by the add-on card is given in the manual provided with the card.

A driver is the software utility that enables a computer to work with a particular device. Devices such as mouse devices, disk drives, DA&C adapter cards, network cards and printers all come with their own driver. The computer's operating system will not recognize a device until its associated driver has been installed, unless the operating system is Plug-and-Play compliant. Drivers are included as a disk with the DA&C card when it is sold.

6 Output Signal Conditioning

Table 2 provides the list of control elements that require digital signals and sensors that give outputs as digital signals. Table 3 gives a list of typical control system actuations. The electrical output voltage and current rating for most DA&C boards is limited to modest electrical ratings. Output ratings are 1.7 mA (sinking) while maintaining 0.45 V and 200 μA (sourcing) while holding the output voltage at 2.4 V. These current and voltage values are satisfactory for interfacing to TTL-compatible devices such as integrated circuit chips. However, when integrating a DA&C board to real-world equipment such as electrical motors connected to pumps, fans and other rotating devices, electromechanical solenoids and other heavy duty electromechanical devices, the TTL-compatible outputs simply cannot drive these devices.

TTL or CMOS I/O
Optically isolated I/O
Electromechanical relays
Solid state relays
Frequency input
Proximity sensors
Photo-electrode sensors
Switches/contacts
Encoder

Table 2 Digital input/output

Control action	Devices	Input
Electric heating	Electric furnace	Voltage or current
Flow adjustment	Pneumatic valve	Air pressure
	Solenoid valve	Voltage, current
	Motor driven valve	Digital pulses
	Variable speed pump	Voltage or current
	Fluidic control valve	Pressure
Alarms	Lights, bells	Digital pulse
On-off signals	Relays, switches	Digital pulse

Table 3 Some typical control system actuations

The output port is said to be sourcing current if current flows out of the port into the load. It is said to be sinking if the current flows through the load and into the output port. Most actuators require more than 100 mW. Since most types of digital logic circuits source or sink 20 mA (100 mW), the ports by themselves are not enough to drive the actuators. The TTL gate will source 16 mA (i.e., supply a current to ground through a load connected to a positive supply) and is turned on by a logic zero. Therefore, discrete circuits are required. Driving circuits are (i) integrated circuits, (ii) discrete solid state devices, (iii) electromechanical relays and (v) solid state relays.

6.1 Electromechanical relays

Electromechanical relays are the primary interface between the solid state electrical current and heavy real-world devices such as motors, pumps and solenoids. If a relay is of high current or voltage rating, then a transmitter must be selected so that the current and voltage ratings are compatible. A transistor is a solid state DC switching device used with low voltage DC-powered conductive and capacitive sensors as the output switch. Advantages of the transistor are instantaneous response, low off-state leakage and voltage drop, longer life and immunity against shock and vibration. Disadvantages of the transistor are low current handling capacity, inability to handle inrush current unless clamped, and change of being destroyed by short circuit unless protected. Proper buffering and current are needed for an interface between the TTL level and the real world of electromechanical relays. A contact can be either normally open or normally closed. The current and voltage requirements of relays are more than the TTL gate can provide. Transistor buffers can be used to drive a relay coil. The contact life depends on the load current and frequency of operation. The relay coil must be shunted by a freewheeling diode to protect it from damage when the driver is turned off. The advantages of electromechanical relays are switching high current loads, multiple contacts, switching AC or DC voltages and tolerance of inrush current. The disadvantages are slow response time (10 to 25 ms), mechanical wear, corrosion of contacts due to arcing, contact bounce and vulnerability to shocks and vibration.

6.2 Solid state relay

The problems of relays can be overcome by replacing relays by solid-state relays (SSRs) such as a triac or silicon-controlled rectifier (SCR). SSRs are capable of controlling only AC operated devices such as motors or heaters. The advantages are fast response time (8 ms) and longer life. The disadvantages are that the SSR can be falsely triggered by large inductive currents, can be destroyed by short circuit, and it is used only for controlling AC.

An optocoupler (or optoisolator) is a device that provides electrical isolation between a source and a load. The optocoupler consists of a light emitting diode (LED) in the input and a photo sensor at the output with no electrical connection between them. The photo sensor may be a photo transmitter, or an SCR, photo Darlington type or photo triac type. SCRs or triacs are suitable for AC power switching applications (such as motors and solenoids) whereas the Darlington output is used for applications requiring large current gains and the photo transmitter output for general-purpose applications. Optocouplers allow interfacing systems with different grounds, etc., which would otherwise be incompatible with the controllers.

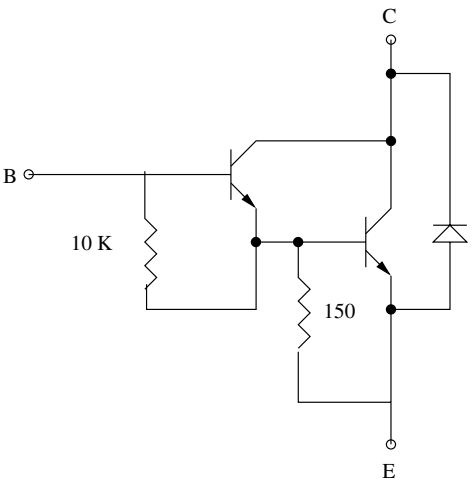


Fig. 11. Power Darlington solid state device

The current gain of the transmitter switch is improved by using the Darlington connection (refer to Fig. 11). This has two (or more) transistors wired so that the base current drive to the output transmitter is supplied from the emitter of the first. This type of connection gives a very high value of current gain (typically 200 to 1000). The Darlington device can switch a large output current (2 A) with a relatively small input current (mA). In Fig. 11, two resistors and a diode are included to reduce leakage current. An output load

power of, say, 300 W can be switched with an input power of about 60 mW. Since the collector to emitter current is higher, resulting in a high amount of heat, a proper air cooling arrangement must be made.

6.3 I/P converter

If the final control element is a pneumatic operated valve, then the output signal to the valve has to be converted into a pneumatic signal by using an I/P converter. Here, depending on the current (4–20 mA) from the controller, the output of an I/P converter is a pneumatic signal which will be sent to the pneumatic valve. A constant air supply is given to the I/P converter which will send an appropriate amount of pneumatic signal to the valve and bleed out the remaining portion of the air signal. The valve will be opened or closed proportional to the signal obtained from the I/P converter.

6.4 Heater control circuit

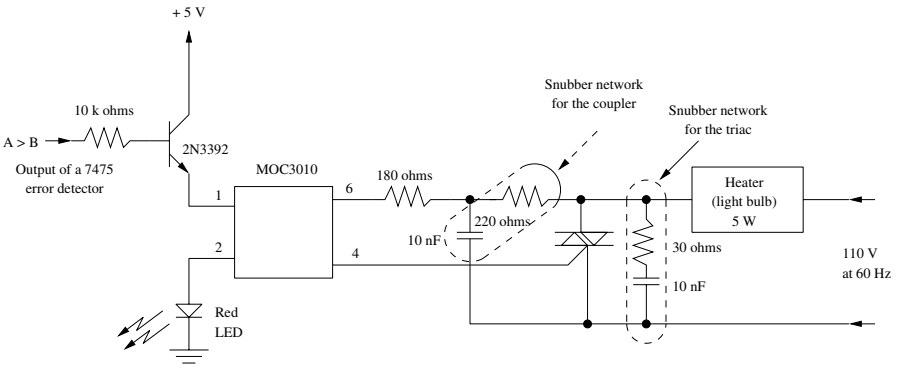


Fig. 12. Drive circuit for the heater (or light bulb)

If the desired temperature is higher than the actual temperature (output A > B), then the heater is to be on (refer to Fig. 12). We also isolate the high voltage AC signal from the low voltage digital signal from the computer. An optically isolated drive MOC3010 satisfies the need. To increase the drive capability of an MOC3010, an additional triac may be connected at the output of the MOC3010 depending on the electrical ratings of the load it drives. The current from the computer may not be sufficient to drive the MOC3010, and then we need current amplification. The schematic diagram of the amplifier, optically isolated triac driver and a triac and a snubber circuit are shown in Fig. 12.

6.5 Need for a snubber circuit

To prevent turn-on due to overvoltage or line transients, a varistor is connected in parallel with the SCR or triac. This also prevents the SCR from being damaged by overvoltage during its off state. If the voltage applied to the anode changes very abruptly, the SCR can turn on and short conducting (this is due to the effect of rapidly changing voltage upon the SCR's junction capacitance). To reduce the effect, a combination series resistor/capacitor network is connected in parallel with the SCR. The parallel branch is called a snubber. The snubber helps in the following ways: (i) The parallel capacity appears as a temporary conduction circuit to a rapidly changing anode voltage. This circuit is parallel with the internal junction capacitance, and if the external capacitor is of the correct size, it virtually shorts the SCR during the period of rapid change. Thus the false turn-on is eliminated. (ii) When the SCR is conducting an inductive circuit, the rise of current in the circuit is delayed due to the effects of the inductance. Thus the growth of the conducting surface area, within the SCR, to the application of the trigger pulse and the flow of load current is limited, and the energy contained within the pulse is not sufficient to establish conductance. However, when a snubber network is present, the external capacitor will probably already be charged, and at the time at which the firing pulse is applied, the capacitor discharges via the SCR and aids in establishing current flow.

Basically, two types of solid state relays are commercially available: zero-cross switching and random switching. Zero-cross switching relays do not switch on until the first zero-crossing of the line voltage after the control signal is applied. On the other hand, random switching relays switch on immediately after receipt of the control signal. A proper type of device should be selected, depending on the nature of the application.

6.6 Pulse width modulated (PWM) amplifiers

In a PWM motor control, the DC voltage is switching rapidly across the armature, and the current through the motor is affected by the motor inductance and resistance. Since the switching speed is high (the frequency is often in excess of 1 kHz), the resulting current through the motor has a small fluctuation around the average value. The duty cycle is defined as the ratio between the on time and the period of the waveform, usually specified as a percentage. As the duty cycle becomes larger, the average current becomes larger and the motor speed increases.

7 Cabling

Several types of cables are commonly used for interfacing tasks. In increasing order of cost and immunity from noise, the choices are single cables, flat cables,

twisted pair conductors, coaxial cables and triaxial cables. Flat cables consist of many thin conductors running parallel to one another and are insulated with plastic. The flat cable has almost the same characteristics as single wire but allows the cabling to be more organized. Both of these cablings are suitable when the signal is 1 to 10 volts and the current is less than 100 mA (i.e., low noise situation). For a flat cable, it is easier to use a single connector. Many analog I/O plug-in cards for the IBM PC use 25 conductor flat cable for analog and digital signal connections.

Any signal that interferes with the signal of interest is called an interference or, more commonly, a noise. Electrical noise refers to currents induced by coupling external electric fields and the wiring in the instrument. Most of the methods for dealing with electrical noise are based on the principles of the Faraday cup, which is a big conducting cup (or cage) that surrounds the object of interest. The cup is connected to the ground. The objects inside the Faraday cup are effectively isolated from external electrical fields. Magnetic noise refers to the current in the wiring of the instruments when they are placed near a changing external magnetic field. The usual technique with magnetic noise is to physically isolate the instruments from large alternating magnetic fields such as those found in electric motors. For either type of noise, the smaller the level of the signal of interest, the more important is the noise prevention.

Twisted, shielded pair wiring provides a very effective shield against electrostatic and magnetic coupling. The twisted pair conductors can be used for differential, as opposed to single ended, analog (A/D, D/A) I/O connections. Twisted pair wire normally consists of four or eight copper strands of wire, individually insulated around each other in braided pairs and bound together in another layer of plastic insulation (they were originally just two wires, rather than four or eight, hence the name *twisted pair*). Except for the plastic coating, nothing shields this type of wire from outside interference, so it is called unshielded twisted pair (UTP) wire. Some twisted pairs of wire are further encased in a metal sheath and this setup is called shielded twisted pair (STP) wire. Twisted pair wire is more economical than coaxial cable. Twisted pair wire can be used where greater bandwidth and greater noise reduction than those provided by flat cables are necessary.

A coaxial cable consists of two conductors: one is a single wire in the center of the cable and the other is a wire mesh shield that surrounds the first with an insulator in between. Current flows in one direction on the inner conductor and in the opposite direction on the outer conductor. As a result of this bidirectional current flow, the electromagnetic fields generated within the two conductors cancel each other. High frequency analog signals may be transmitted over coaxial cables without causing any interference with other laboratory electronic instruments, and with little loss of signal. The coaxial (or coax) cables are shielded so that laboratory electronic noise does not interfere with the coaxial cable transmission. Bayonet Neill Concelman (BNC) connectors are usually used with the coax cable to provide convenient connections. Impedance matching connectors and terminators are available to improve the

signal quality at high frequencies on coax cables. Coax cables can carry more data than the older types of twisted pair wiring and are less susceptible to interference from other wiring. They are more expensive and have become less popular as twisted pair wiring technology has improved.

7.1 Terminal stripper connectors

Many commercial PC plug-in peripherals for analog data acquisition have strips connecting all the signals of interest. Screw mounts on these strips are provided so that bare wires may be interfaced to the ADC or DAC. Each screw is isolated with a plastic barrier to prevent short-circuiting. The terminal strip connector is mounted on a printed circuit board (PCB) which usually has a shielded flat cable connecting the PCB with the D-type connector of the PC data acquisition plug-in card. Terminal strip connectors thus provide convenient connections, but may provide relatively little signal isolation, depending upon the cable attached to the terminal strip.

Digital signals use two voltage levels (or frequencies) that are widely separated from one another to do their signaling. Hence, even rather large amounts of noise result in little degradation in the ability to distinguish between these two signal levels. As a result, almost all digital systems use a cheap cable; the most common are flat cable for parallel digital I/O systems and twisted pair for serial digital I/O systems. At very high frequencies, the noise may be more of a problem, and shielded wires are required.

8 Process Control Example for DA&C

Fig. 13 shows the flowsheet of a pH control using a computer. The effluent stream from chemical industries usually will be acidic in nature. The stream has to be neutralized by adding appropriate amounts of alkali (sodium hydroxide solution) in a mixing tank. The pH of the solution in the mixing tank is measured using a pH electrode. The output of the electrode will be in the millivolt range. This weak signal has to be signal conditioned (amplified to 0–10 V). Since the conditioned signal has to be sent a long distance where a PC is kept (maybe in an air-conditioned room), there is a need to convert the 0–10 V signal to a current (4–20 mA) signal. The signal requirement for an add-on card is 0–10 V. Hence this current signal has to be converted into a 0–10 V signal. In the PC, appropriate control calculation will be carried out (PID control action or nonlinear control action). The calculated signal from the DAC in the add-on card will be 0–10 V. For transmission of the signal to the final control element, the signal will be converted into 4–10 mA. At the control valve side, this signal will be converted to a pneumatic signal (3–15 psig) using an I/P converter for actuating the pneumatic valve. An instrument air supply should be available to the I/P converter.

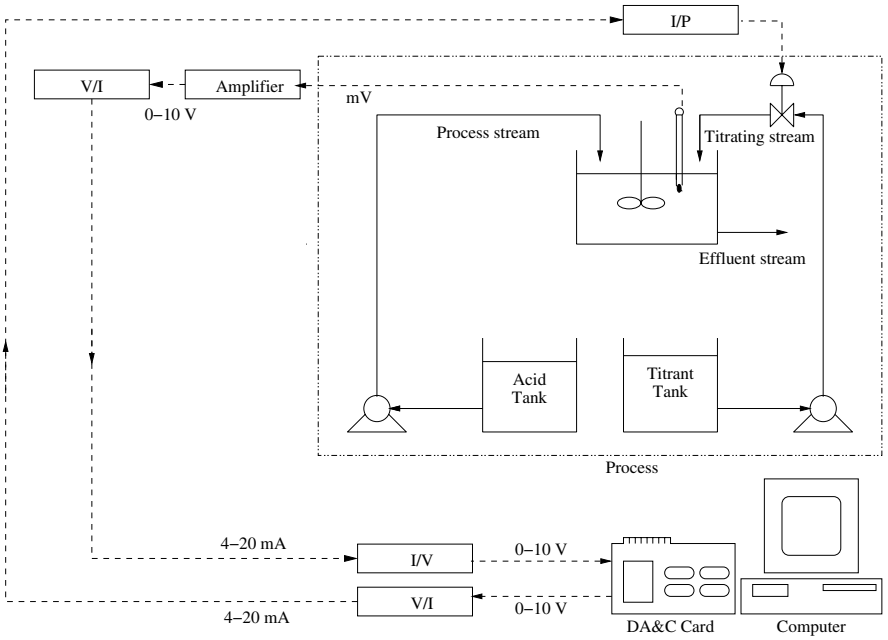


Fig. 13. Flowsheet of a pH control

9 VXI Standard

In the late 1980s, a new VXI standard was introduced, which permits communications with a transfer rate of 20 megabytes per second between VXI systems. VXI instruments are installed in a rack and are controlled by and communicate directly with a VXI computer. These instruments do not have buttons or switches for direct local control and do not have a local display. They can be used for compact monitoring systems.

As stated earlier, the sensors typically require some sort of signal conditioning. Various choices are available in the packaging of sensors and signal conditioners. These include (i) data loggers, (ii) PC plug-ins, (iii) computer backplane PC plug-ins and (iv) network-based systems. A data logger solution is often a stand-alone instrument but could include a low cost interface to the personal computer. A PC-based plug-in solution generally uses available low cost slots in the computer to hold measurement modules. A computer backplane solution makes use of a separate main frame holding multiple modules connected to a personal computer with a high speed interface. Their strength comes from being able to accommodate large channel counts, provide an extremely wide range of measurements and record/analyze the resultant data with high real-time speeds. A network-based solution can be physically distributed, inheriting the advantages of local area network (LAN) technology. Using such standards as the Ethernet, small measurement modules can be

connected to PCs, sometimes at greater distances. This new architecture provides an information bandwidth greater than that typically available from a standard data logger, but less than that available from a computer parallel-backplane architecture. The network-based system encourages digitization of the measurement data as close to the point of measurement as possible, then brings the data to the controller digitally.

10 Software

Every computer language offers an input instruction that enables the computer to receive data from sensors or switches via an internal board. The language also provides an output instruction that allows the computer to output data to the external world through an interface board connected to the computer. Let us discuss the application of the language BASIC to I/O applications. IBM computers and IBM clones use INP (input) and OUT (output) statements, whereas the computers that use Motorola processors such as the Apple Macintosh use PEEK (input) and POKE (output) I/O statements.

The form of the input instruction (INP) in the BASIC language is given by INP(n), where n is an integer which represents a number or port address of the input port being selected or accessed through the software. The statement
Indata = INP(&H300)
contains a variable name assigned to the data and that data has been input to the computer via the address port. The input port address is provided in hex (hexadecimal). The ampersand (&) used in this statement indicates that the port address within the parentheses is provided as a hexadecimal number. In contrast, in the statement
Indata = INP(768)
the port address is to be given by a decimal equivalent of the port address.

Output from the computer through an I/O port is initiated by the out statement. The form of the out statement in BASIC is given by
OUT port, byte
In this statement, (port) is the output port address in hex or decimal form and (byte) is the data in hex, decimal or binary form that is going to be output through the specified port. The data byte output to a port must be eight bits. Even if not all of the eight bits are used in the output circuitry, the data byte sent to the port must be a full byte. In the statement
OUT &H301, &HFF
the output port address on the I/O board is 301 hex and the digital data delivered to the port consists of eight ones (11111111). The two statements
LET LEDS = &HF0
OUT &H301, LEDS
will cause a binary (11110000) to be delivered to output port 301 hex. Assuming that the LEDs were connected to this port, four LEDs would be lit and four LEDs would be off.

The Blue Chip Technology ACM-42 DA&C card has a factory-set base address of &H300. The user can alter this base address, if required, by the adjustment of miniature jumper switches on the card. All ports used by a controlling program are referenced to the base address as follows:

Base+0 ADC bit 7 going high registers an end of conversion signal.

Base+1 ADC result, high byte with four most significant bits set to zero.

Base+2 ADC result, low byte and automatic start conversion signal.

Base+3 DAC update output.

Base+4 DAC A, low byte load register.

Base+5 DAC A, high byte load register.

Base+6 DAC B, low byte load register.

Base+7 DAC B, high byte load register.

Base+8 Digital I/O port A.

Base+9 Digital I/O port B.

Base+10 Digital I/O port C.

Base+11 Digital I/O control register.

Base+12 Analog multiplexer channel select.

Base+13 Programmable interrupt source control.

If the input ADC or output DAC facilities are used, they can be accessed using the relevant base offsets given above. The analog input is set for bipolar inputs in the range of −2.5V to +2.5V, but this can be changed, if necessary, by adjusting the jumper connections on the card. If the digital I/O ports are to be used in any application however, then the control register, Base+11, must be set up as required by writing the appropriate control word to the control register. The instruction OUT BASE+11, &H93, for example, configures the port A, port B and the upper four bits of port C set for input. The lower four bits of port C are set for output. The instruction OUT BASE+11, &H80 configures the ports A, B, and C all as output.

The general data acquisition procedure is as follows:

1. Define the card base address with a suitable variable.
2. Select an input channel.
3. Send out a start conversion signal.
4. Check for end of conversion.
5. Read the ADC output.
6. Store data in memory.

A suitable BASIC program to continuously monitor analog input channel 5 is shown below (bipolar range is considered). A heater is to be switched on or off depending on whether the measured signal is higher or lower than the set point.

Program 1. single channel input

```
PORT=&H300                          :REM set base address
OUT PORT+12,5                       :REM select channel 5
OUT PORT+11, &H80                   :REM setting ports A, B, and C as
                                    outputs
INPUT volts set point;SP
OUT PORT, &H00                      :REM heater off
S=INP(PORT+2)                       :REM start conversion by reading
                                    low byte
FOR J=1 TO 10, NEXT J               :REM small time delay to allow for
                                    A/D conversion
CYCLE:
HIGH=INP(PORT+1)                    :REM read high byte
LOW=INP(PORT+2)                     :REM read low byte
RESULT=LOW+256*HIGH                 :REM calculate the 12 bit result
PRINT RESULT
VOLTS=(RESULT-2048)*5/4095          :REM convert digital result to
                                    voltage level
PRINT VOLTS
IF VOLTS >=SP THEN 1000
OUT PORT+8, &H00                    :REM heater off
GO TO CYCLE
1000
```

```
OUT PORT+8, &H01                :REM heater on
GO TO CYCLE
```

In the above program, REM is the keyword in the BASIC language for writing remarks within the program. It is used for understanding the program.

11 Graphical Programming

PC-based monitoring and control systems using graphical programming are used frequently. This permits the utilization of graphical components for assembling systems and simplifies the development of monitoring and control programs so that scientists and engineers as well as nonspecialists in computers can easily develop PC-based monitoring or control systems. The intuitive nature of the graphical programming languages reduces significantly the time required for learning, programming of the prototype and realization of the final product. Textual programming for computer-based instrumentation however has some specific advantages: the user familiarity with a computer aided package like MATLAB allows easy adoption of textual programming for the MathWorks data acquisition tool box. Moreover, in applications requiring the computation of measurement uncertainty, textual programming can be advantageous due to the explicit instruction used in data processing.

Formerly, most of the PC software was DOS-based. Today almost all PC software is designed to run under windows. Most add-on card manufacturers offer their own windows-based software for controlling their cards. This software is adequate for basic I/O functions common to open loop applications but still relies on a conventional text language (BASIC or C) for closed loop applications. An important limitation of this type of software is that it works only on cards provided by the manufacturer, making multi-card, multi brand applications difficult. Packages like LabView, Matrix, Simulink, and VisSim are capable of both open loop and closed loop applications.

12 Industrial Signal Conditioners

Nowadays most manufacturers of electronic instrumentation devices are producing signal conditioners intended to simplify the use of transducers. These signal conditioners have built-in (i) a power supply for passive transducers, (ii) capability to amplify the output signal up to the levels of several volts, directly compatible with the industrial ADCs and (iii) capability to filter either industrial noise or unwanted spectral components in the phenomena measured.

As an example, let us give the family of signal conditioners 5B or 6B from the supplier Analog Devices, including conditioners for strain gauges, thermocouples (with cold-junction compensation) or metallic resistance temperature

detectors (taking into account the nonlinearity of the R-T transducers). These modules are rather compact sealed hybrid circuits (size: 57x57x16 mm); they can be factory adjusted for many different standard measurement ranges and even for customized ranges. They have a rated precision of 0.05% for the temperature range -25 to 800^{o}C and a galvanic insulation up to 1500 V between input and output. The cold-junction compensation of the thermocouple is realized by small auxiliary hybrid circuits and the linearization of resistance temperature detector (RTD) is implemented in the signal conditioner itself. Several modules can be plugged on a printed circuit backplane, and their output is connected in series with an internal analog switch which allows multiplexing the outputs without the need for additional components.

Another example is the hybrid circuit IB32 for the signal conditioning of resistive transducers in a bridge configuration. The circuit features an adjustable excitation voltage for the bridge, an amplifier for the output voltage with a gain ranging from 100 to 5000, a low pass filter, a CMRR of 140 dB, a linearity of 0.005% and a global precision of 140 bits.

SCXI: Signal Conditioning eXtensions for Instrumentation (SCXI) is a high performance, multichannel signal conditioning and data acquisition system supplied by National Instruments for use with PCI. We can use the SCXI as either a signal conditioning front end with DA&C boards and modules, or as a complex external system. An SCXI system consists of one or more rugged chassis that can house a variety of signal conditioning modules for most I/O needs, such as the following:

- *Analog inputs:* Thermocouples, RTDs and thermistors, strain gauges, voltage sources, 4–20 mA current sources, frequency inputs. Analog outputs: voltage and current.
- *Digital I/O:* Optically isolated I/O, AC/DC inputs, solid state relays, electromechanical relays.

Analog input modules interface the system to a variety of transducers and signal conditioning circuits such as those for signal amplification, isolation, multiplexing, filtering, transducer excitation, and simultaneous S&H. For local DA&C systems, we can use the SCXI module to consider I/O signals for plug-in DA&C cards. We can also use the external with the DA&C module to digitize the data to the PC via the parallel port or serial port over a distance of 4000 ft. We can use the SCXI as a remote on RS-486 networks up to 1.2 km from the host PC. The expansion system can range from a few channels for desktop computers to a large rack-mount system with up to 3000 channels.

13 General-Purpose Interface Bus (GPIB)

The GPIB, also known as an IEEE-488 bus, is a system that allows interconnection of up to 15 electronic instruments or devices so that they can interact with each other. There are three categories of devices on the GPIB: talkers,

listeners and controllers. The system is programmable and so can form the basis of automatic test equipment (ATE) systems. ATE is now one of the leading methods for testing electronic equipment in factory production and troubleshooting situations. The data acquisition system can also be designed around standard ATE modules and equipment. The usual method is to use a programmable digital computer to control a bank of test instruments. The program turns the various instruments on and off and then evaluates the results as measured by other instruments. The bank of equipment can be configured for a special purpose or for general use. For example, we can select a particular lineup of equipment needed to test, say, a broadcast audio console, and provide a computer program to make the various measurements: gain, frequency response, total harmonics distortion, etc. Alternatively, we can also make a generalized test set. This is the method selected by a number of organizations that have numerous different devices to test. There will be a main bank of electronic test equipment, adapters to make the devices under test interconnect with the system, and a special program for each type of equipment. Such an approach provides a cost effective system of test equipment. Nowadays, electronic test equipments are either fitted with the necessary GPIB interface or can be upgraded with optional GPIB interface cards. A GPIB card installed in a PC would make it a GPIB device, and it connects the adapter card to the instruments using a special GPIB cable.

14 Microcontrollers

Perhaps more than any other factor, the development of microprocessors has been responsible for the explosive growth of the computer industry. A microprocessor is a computer on a chip. While early microprocessors required many additional components in order to perform any useful work, the increasing use of large-scale integration (LSI) or very large-scale integration (VLSI) semiconductor fabrication techniques has led to the production of microcomputers, where all of the required circuitry is embedded on one or a small number of integrated circuits. A further extension of the integration is the single chip microcontroller, which adds analog and binary I/O, timers, and counters so as to be able to carry out real-time control functions with almost no additional hardware. Examples of such microcontrollers are Intel 8051, 8096, and Motorola MCH 68HC11. These chips were developed largely in response to the automotive industries' desire for computer-controlled ignition, emission control and antiskid systems. They are now widely used in process industries. The major types of microcontrollers are (i) embedded 8-bit microcontrollers, (ii) 16- to 32-bit microcontrollers and (iii) digital signal processors.

The basic characteristics of a microcontroller are as follows:

1. It has a built-in ROM (4 Kbytes) within the chip to store the control program.

2. It has a small built-in RAM (128 bytes) for temporary data storage.
3. The CPU uses single bit instructions so that the limited program memory (ROM) is effectively used.
4. Many microcontrollers have a Boolean co-processor along with the CPU to simplify implementing Boolean expressions occurring often in control applications.
5. Microcontrollers have built-in counters and timers which can be set by users.
6. There are built-in I/O ports and control for easy interaction with external devices.

Digital signal processing is defined as the arithmetic processing of signals sampled at regular intervals. Examples of this type of processing are filtering, convolution, amplification, modulation and transformation of signals. A digital signal processor (DSP) basically replaces analog controllers and conventional microprocessors in digital control system applications. Because of the DSP's special architecture, it is more useful than a general-purpose microprocessor for high speed processing applications. Because a control system is a real-time system, the DSP architecture must handle a control system's numerical tasks and band width requirements. Microcontrollers traditionally have a von Neumann architecture (meaning that instruction and data are in the same bus). However, most DSPs use a Harvard architecture (meaning that instruction and data are separated from the instruction bus to increase the speed) or a modified Harvard architecture that is optimized for signal processing.

15 Summary

Computer interfacing for data acquisition consists of analog-to-digital conversion of input analog signals. Prior to the conversion, the analog signal has to be conditioned to meet the input requirements of the ADC. Signal conditioning consists of amplification (for sensors generating very low level signals), filtering (to limit the amount of noise on the signal) and isolation (to protect the sensors from interacting with one another and/or to protect the signal from possible damaging inputs). Conversion of a digital signal to an analog signal at the output is to be carried out if the output signal is sent to a final control element which requires an analog signal. The digital output signal has to be amplified by a transistor or solid state relay or power amplifier. DA&C can be made simple by using a PC's standard add-on card and associated software. Software configurable cards with auto isolation devices are preferable. Most manufacturers of electronic instrumentation devices are producing signal conditioners as modules.

References

1. Tompkins, W.J. and I.G. Webster (1988), *Interfacing Sensors to IBM PC*, Prentice-Hall, Englewood Cliffs, NJ.

2. Gates, S.C. and J. Becker (1989), *Laboratory Automation Using the IBM PC*, Prentice-Hall, Englewood Cliffs, NJ.
3. Carr, J.J. (1991), *Micro Computer Interfacing*, Prentice-Hall, Englewood Cliffs, NJ.
4. Gupta, S. and J.P. Gupta (1994), *PC Interfacing for Data Acquisition and Process Control*, 2nd Ed., ISA, Research Triangle Park, NC.
5. Rigby, W.H. and T. Dalby (1995), *Computer Interfacing: A Practical Approach to Data Acquisition and Control*, Prentice-Hall, Englewood Cliffs, NJ.
6. Tooley, M. (1995), *PC-Based Instrumentation and Control*, 2nd Ed., Newnes, Oxford.
7. Young, S.S. (2001), *Computerized Data Acquisition and Analysis for the Life Sciences*, Cambridge University Press, London.
8. Olsson, G. and G. Piani (1992), *Computer Systems for Automation and Control*, Prentice-Hall International, London.
9. Shetty, D. and R. Kolk (1997), *Mechatronics System Design*, PWS Publishing Company, Boston, MA.
10 Frasher, C. and J. Milne (1996), *Electro Mechanical Engineering: An Integrated Approach*, IEEE Press, New York.
11. Kilian, C.T. (2001), *Modern Control Technology: Components and Systems*, 2nd Ed., Delmar Pub, Singapore.
12. Necsulescu, D. (2002), *Mechatronics*, Pearson Education Pvt. Ltd., Singapore.
13. Chidambaram, M. (2002), *Computer Control of Processes*, Narosa Pub, New Delhi.

Programmable Logic Controllers

Gustaf Olsson

Lund University, Lund, Sweden
gustaf.olsson@iea.lth.se

A computer system for automation has to satisfy many requirements that we take more or less for granted. It has to run around the clock since a production break may cost enormous amounts of money. It has to work in real time taking various time requirements and process disturbances into consideration. Whatever happens, the system has to behave in a predictable way. It has to be safe, both for the process and for humans.

In most automation systems there are events that will bring the process into another state of operation. Furthermore, there are a lot of applications in both the process and manufacturing industries where control involves primarily switching and sequencing. In both the process and manufacturing industries there is a wealth of applications of switching circuits for combinatorial and sequencing control.

Switching theory, which provides the foundation for binary control, is not only used in automation technology but is also of fundamental importance in many other fields. This theory provides the very principle on which the function of digital computers is based. In general, binary combinatorial and sequencing control is simpler than conventional feedback (analog and digital) control, because both the measurement values and the control signals are binary. However, binary control also has specific properties that have to be considered in more detail.

Programmable logical controllers (PLCs) have been in use since the 1960s and are still the basis for the low level control in many automation systems. Today PLCs can handle not only the lowest levels of control but also advanced control of hybrid systems, where time-driven continuous controllers have to be integrated with event-driven controllers. We will introduce the PLC and briefly review its short history. The state concept is of fundamental importance in understanding sequencing control. This will be described by a simple example, followed by an introduction to sequential function charts (SFCs). They can be used not only for simple control sequences but also for parallel processes. Ladder diagrams (LDs) are inherited from the old electromechanical relays and are nowadays implemented in software in PLCs. Another low level

description of combinatorial circuits and sequences is obtained via assembly-like Instruction Lists (ILs). Programming of a modern PLC can be realized in five different programming languages, text oriented or graphical. An international standard, IEC 61131-3, forms the basis for advanced automation system programming [6]. There is a wealth of literature on PLCs and their applications. There are several books with good coverage on, not only PLCs and the IEC 61131 standard, but their application in discrete manufacturing (see [1]–[5]).

1 The Development of PLCs

The modern computer control system of today is the result of two parallel developments, one from relay technology to implement logical circuits and the other from continuous instrumentation and pneumatic proportional-integral-derivative (PID) controllers developing into software realizations of continuous controllers.

Logical circuits have traditionally been implemented with different techniques. The primary reason for designing a PLC was to eliminate the large cost involved in replacing the complicated relay-based machine control systems. When production requirements changed, so did the control system. This becomes very expensive when the changes are frequent. Since relays are mechanical devices they also have a limited lifetime, which required strict adhesion to maintenance schedules. Troubleshooting was also quite tedious when so many relays were involved. Picture a machine control panel that included many, possibly hundreds or thousands of, individual relays. Then it is easily recognized that alternative solutions were sought.

Bedford Associates (Bedford, MA) proposed something called a Modular Digital Controller (Modicon) to the U.S. car manufacturer General Motors. The first PLC was introduced at GM in 1968. Other companies at the time proposed computer-based schemes, one of which was based upon the Digital Equipment PDP-8. The Modicon 084 brought the world's first PLC into commercial production.

These "new controllers" also had to be easily programmed by maintenance and plant engineers. The lifetime had to be long and programming changes easily performed. They also had to survive the harsh industrial environment, both mechanically and electrically. The answers were to use a programming technique most people were already familiar with and replace mechanical parts with solid-state ones. The new device had to be smaller than its relay or semiconductor-built equivalent and it had to be easy to maintain and repair. In addition, the new device had to be cost-competitive with the solid-state and relay panels then in use. These requirements should be considered in the light that at the end of the 1960s and beginning of the 1970s there still were no small-size programmable computers (the microprocessor was invented in 1971). The initial requirements provoked great interest from

engineers of all disciplines as to how the PLC could be used for industrial control. Allen-Bradley Corporation in the USA introduced a microprocessor-based PLC in 1977. It contained an Intel 8080 microprocessor, and additional circuits allowed processing of logical bit operations at high speed.

Still, until the mid-1970s most circuits were built with electromechanical relays and pneumatic components. During the 1970s the PLC became more and more commonplace, and today sequencing control is almost exclusively implemented in software. Despite the change in technology, the symbols for the description of switching operations, known as ladder diagrams, that derive from earlier relay technology are still used to describe and document sequencing control operations implemented in software.

In the mid-1970s the dominant PLC technologies were sequencer state-machines and the bit-slice based CPU. The AMD 2901 and 2903 were quite popular in the Modicon and Allen-Bradley PLCs. Conventional microprocessors lacked the power to quickly solve PLC logic in all but the smallest PLCs. As conventional microprocessors evolved, larger and larger PLCs were based on them. Today there are hundreds of different PLC models on the market, differing by their memory size and number of I/O channels but primarily in the features they offer. The smaller PLCs are designed principally to replace relays and have some additional counting and timing functions. More complex PLCs process analog signals, perform mathematical calculations, and even contain feedback control circuits like PID controllers.

In the process industries, like the paper and pulp, chemical, and oil and gas industries, there has been a parallel development of control systems. The typical control loop consisted of a pneumatic PID controller connected to some analog sensor for a variable like level, pressure, flow rate, or temperature. The controllers were gradually realized by electronic circuits, but the functionality was still the same. More than one controller could soon be implemented into one device.

When Honeywell introduced the TDC 2000 system in the mid-1970s, it was considered a great sensation and a big leap forward. In direct digital control (DDC) all the controllers were implemented in a single computer. Some control engineers considered DDC as the ultimate solution, while the practicing engineers disliked the extreme sensitivity of the system. If the computer failed, then the whole process control system would fail. A distributed computer control would of course minimize the risk of plant failure.

Communications abilities began to appear in approximately 1973. The first such system was Modicon's Modbus. The PLC could now talk to other PLCs and PLCs could be far away from the actual machine they were controlling. They could also now be used to send and receive varying voltages, allowing them to enter the analog world. Unfortunately, a lack of standardization, coupled with continually changing technology, has made PLC communications a nightmare of incompatible protocols and physical networks. Still, the 1970s was a great decade for the PLC.

The 1980s saw an attempt to standardize communications with General Motor's Manufacturing Automation Protocol (MAP). It was also a time for reducing the size of PLCs and making them software programmable through symbolic programming on personal computers instead of dedicated programming terminals or handheld programmers. Today the smallest PLC is about the size of a single control relay, just a few centimeters across.

The 1990s saw a gradual reduction in the introduction of new protocols and the modernization of the physical layers of some of the more popular protocols that survived the 1980s. There is a textbook [9] that tells almost anything that is to be told about computer communication. The Ethernet is being increasingly applied at the automation level, and the Ethernet Industrial Protocol (Ethernet/IP) is an open, industrial Ethernet standard, managed and promoted by several leading industrial network trade associations. A comprehensive description is given in [10]. The fieldbus market is steadily changing; current standards and applications can be found on www.fieldbus.org. A comprehensive description is given in [11].

The latest standard, IEC 61131-3, has tried to merge PLC programming languages under one international standard. We now have PLCs that are programmable in ladder diagrams (LDs), function block diagrams (FBDs), instruction lists (ILs), C, and Structured Text (ST) all at the same time. PCs are also being used to replace PLCs in some applications.

The PLC systems and the DDC systems grew out of two different industrial needs. In order to understand the different programming system developments that are now integrated into modern systems, we have to realize that two different kinds of technicians served the systems. The relay systems were designed and maintained by electricians, who still wanted to structure the software as the old electromechanical relays. Therefore, the ladder diagram is still a natural way of thinking for many of these technicians. Instrumentation engineers, on the other hand, served the process control loops. They wanted to continue thinking in terms of the PID control loop that had to be implemented in software. These traditions have been implemented very favorably in the IEC standard 61131, allowing the two traditions to be combined into the same computer control system.

The automation industry is highly competitive, and many of the computer control companies from the 1980s and the 1990s have now disappeared. Competing companies have acquired many of them; others have gone out of business. Six large companies, having 84% (in 2002) of the automation market, today dominate the automation market. They are ABB, Honeywell, Invensys, Emerson, Siemens, and Yokogawa. Each one of these had between 9 and 21% of the market share in 2002, while the seventh company, Alstom, had only 3%. The web-based information from the vendors provide valuable information.

2 The Finite State Concept

A discrete event system can be described as always being in one well-defined state. For example, a machine can be *operating* or *idle*, which means that it is always in one of these two states. A buffer storage can be in many states, equal to the number of places (N) plus one. The complexity of a discrete state system is not the sum of all the states, but is closer to the product of all the states, if all the variants have to be described. For example, a system of two machines—each one described only by the two states *operating* and *idle*—is described by four states.

Some condition has to be satisfied in order to transfer between two states. Such a condition can, e.g., be an external event, an operator command, or a timer signal. When a machine operation is finished, its state will change from *operating* to *idle*. In our models we assume that such a state transition takes place immediately. Likewise, a timer can indicate a state transfer, for example, the start of a pump. Consequently, the pump condition will (immediately) change from *idle* to *operating*.

In a discrete event control system there are two basic elements, *states* and *transitions*. While the system is in one state there will be some action (operation) taking place. We will illustrate this by a simple example.

A tank is to be filled with a liquid. When the tank is full its content must be heated up to a predefined temperature. After a specified time, to make sure that the liquid is well mixed, the tank is emptied, and the process starts all over again. Let us now define the states and the state transitions of this process.

- A sensor signal *empty* signals that the tank is empty and can be filled again. This is defined as the *initial* state of the operation.
- A signal *start* will initiate the filling of the tank. The start signal then indicates the transition from the *initial* state to the *filling* state. In this state there are two actions being performed. First the bottom valve of the tank is closed. Then a filling pump is started.
- The next transition signal is a sensor signal, indicating that the tank is full. This transition will bring the tank into the state *heating*. Now there are another two actions being started. First the filling pump is turned off, and then a heater is switched on.
- The tank will remain in the *heating* state and the heater will stay on until the temperature has reached the predefined setpoint. When the preset temperature has been reached another transition takes place. At this point the state will be transferred to the state *wait*.
- In the *wait* state the heater is first switched off and a timer is initiated. The timer will run a predefined waiting time "time out" to make sure that the temperature in the liquid has become homogeneous. A mixer may also be running in this state.
- The timer initiates the next state transition, to the *emptying* state. The action *open the discharge valve* is initiated and will stay on until the tank

is empty. Once the tank is empty a sensor signal will indicate *empty*, and there is a state transition to return to the *initial* state.

We again note some typical features:

- The system is only in one state at a time;
- The state transition is initiated by some sensor, timer, or operator signal and it takes place *immediately*;
- While the system is in one state some action will take place. There may be more than one action at the same time. The actions have to stop at the next state transition.

In order to implement the various states and the state transitions, the software has to guarantee that the three conditions above are satisfied at all times.

3 Describing States and Transitions Using Sequential Function Charts

A sequential function chart (SFC) can be considered as a special-purpose language for the description of control sequences in the form of a graphical scheme. Toward the late 1970s the first function chart language, Grafcet (GRAphe de Commande Etape-Transition, "Function chart-step transition") was developed in France and later provided the basis for the definition of the international standard IEC 848 (Preparation of function charts for control systems).[1] Now there is an international standard for the control of sequences, IEC 61131-3. This standard lists a number of alternative languages. Of these, the SFC is the most important. It may be noted that the IEC 61131-3 standard does not really consider an SFC to be a programming language, but rather a *program-structuring* element used to organize the program written in one or more of the other languages. Here we will consider the SFC a programming language.

Function charts describe control sequences with the help of predefined rules for:

- The controls that must be carried out and the order in which they are carried out;
- The execution details of each instruction.

The function diagram is correspondingly divided in two parts (Fig. 1). The "sequence" part describes the order between the major control steps. It consists of the states (marked by the five boxes to the left), also called *steps* in the SFC. The vertical lines that connect each box with the following one

[1]The International Electrotechnical Commission (IEC) has the web page http://www.iec.ch.

represent active connections (directed links). Each transition from a step to the following one is connected with a logical condition, the transition condition or receptivity. The Boolean expression for the transition condition is written in proximity of a small horizontal line that intersects the link from one box to the next. When the logical condition is satisfied, i.e., the related Boolean expression is true, the transition takes place, and the system proceeds with the following step. The actions taking place in each state (step) are described by the "object" or "control" part of the diagram. This part consists of the boxes to the right of the sequence steps. Every action has to be connected to a step and can be described either by an LD, a logical circuit, a Boolean expression, or even a continuous control action like a PID controller.

The use of function charts will now be illustrated with the batch tank example of the previous section. The states can now be recognized in Fig. 1 as the boxes numbered 1–5, while the state transitions are the marked signals between the states.

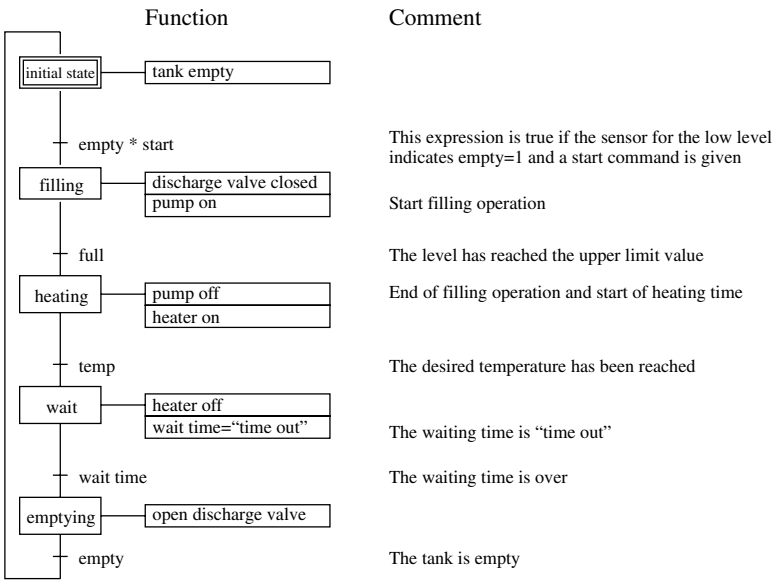


Fig. 1. SFC for the control of a batch tank process

In the function charts syntax a step (= state) at any given time can be either active or inactive. "Active" means that this step is currently being executed. The initial step is represented in the function chart by a double-framed box. An "action" is a description of the commands that have to be executed at each step. A logical condition can be associated with a step, so that the related commands are executed only when the step is active and the condition is fulfilled. Therefore, the association with a condition represents

a security control. Several commands can be associated with a step. These commands can be simple controls but also represent more complex functions like timer, counters, regulators, filtering procedures, or commands for the external communication.

The function chart syntax allows much more than just the iterative execution of the same control instructions. The three functional blocks initial step, step(s), and transitions can be interconnected in many different ways, thus allowing the description of a large number of complex functions. Three types of combinations are possible—in analogy with Petri nets:

- Simple sequences;
- Execution branching (alternative parallel sequence);
- Execution splitting (simultaneous parallel sequence).

In the simple sequence there is only one transition after a step and only one step after a transition. No branching takes place. In the alternative parallel sequence (see Fig. 2), there are two or more transitions after one step. In this way the execution flow can take alternative routes depending on external conditions. Often this is an *if-then-else* condition, and it is useful to describe, e.g., alarm situations.

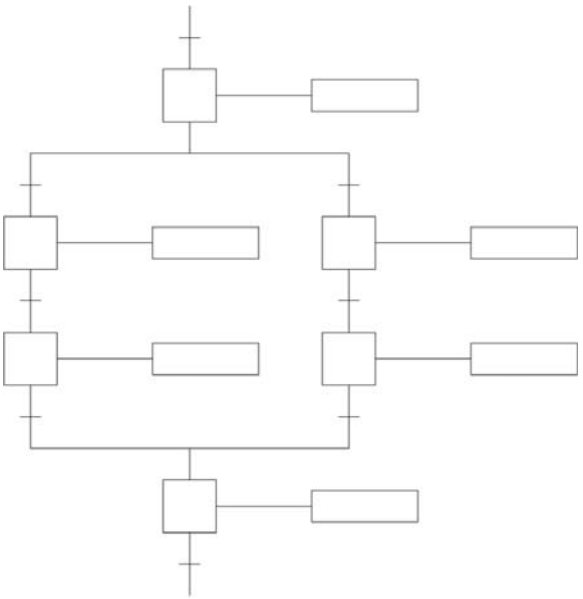


Fig. 2. Alternative parallel paths. The execution is performed from the top and downward. At the branch there is a selection of one out of two alternative execution paths.

In the alternative parallel sequence it is very important to verify that the condition for the selection of one of the program execution branches is consistent and unambiguous; in other words, the alternative branches should not be allowed to start simultaneously. Each branch of an alternative parallel sequence must always start and end with logical conditions for a transition.

In the simultaneous parallel sequence (see Fig. 3), two or more steps are foreseen after a transition, and these steps can be simultaneously active. The simultaneous parallel sequence represents the concurrent (parallel) execution of several actions.

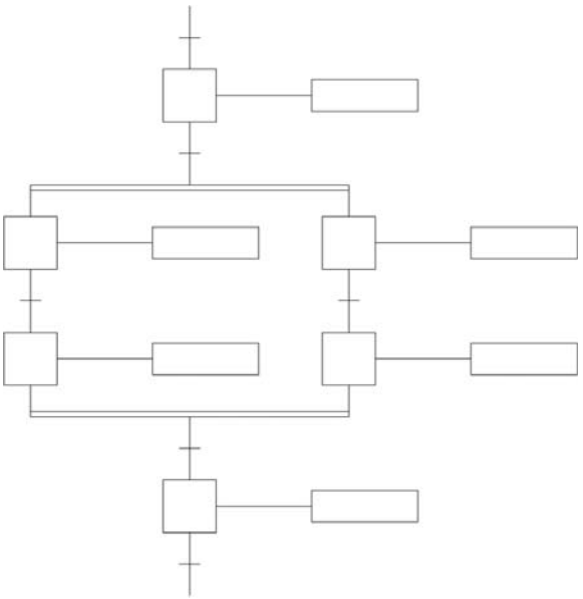


Fig. 3. Simultaneous parallel paths. The execution is performed from the top and downward. At the branch both the concurrent paths are executed simultaneously as two parallel tasks. Before they meet, the fastest branch has to wait for the other one to become completed before the execution can continue.

The double horizontal lines indicate parallel processing. When the condition for the transition is satisfied, both branches become simultaneously active and are executed separately and concurrently. The transition to the step below the lower double horizontal line can take place only after the execution of all concurrent processes has been terminated. This corresponds to the simultaneous execution of control instructions and is comparable with the notation *cobegin-coend*, used in real-time programming.

The three types of sequence processing can also be used together. However, one should act carefully in order to avoid potential conflicts. For example, if two branches of an alternative execution sequence are terminated with the

graphic symbol for the end of parallel execution (the double horizontal bars), then further execution is locked, since the computer waits for both branches to terminate their execution, while only one branch was started because of the alternative condition. Also, the opposite error is possible. If parallel branches that have to be executed simultaneously are terminated with an alternative ending (a single horizontal bar), then many different steps may remain active, so that further process execution might no longer take place in a controlled way. Of course, a compiler would recognize such a mismatch of beginning and end clauses and would thus alarm the user before the code was executed. But even with the best compiler around, many errors remain tricky and undetectable. A structured and methodical approach on the part of the programmer is always an important requirement.

4 Computer Implementation of SFCs

Programs written with the help of functions charts operate under real-time conditions, so each implementation must exhibit real-time capabilities. Usually, the realization of real-time systems requires intensive efforts with considerable investments in time and personnel. However, in this specific case, the designer of the function chart language compiler carries most of the burden, while the user can describe complex control sequences in a comparatively simple way. The aspects of real-time programming are also valid for the design of PLCs, but concern the final user only indirectly and in a limited way.

Compilers for function charts are available for many different industrial control computers. The programming and program compilation on PCs is commonplace. After compilation the code in the form of control instructions is transferred to a PLC for execution. The PC is then no longer necessary during the real-time PLC operation. Some compilers can also operate as simulation tools and show the execution flow on the computer screen without needing to be connected to the object PLC. There are also PLCs with the compiler already built into their software.

The obvious advantage of abstract descriptions in the form of function charts is their independence from any specific hardware and their orientation to the task to be performed rather than to the computer. Unfortunately, it must be said that high level languages like function charts do not yet enjoy the success they deserve. It seems odd that so many programmers always start anew with programming in low level languages, even for those applications that would be much easier to solve with function chart description languages.

As in any complex system description, the diagram or the code has to be structured suitably. A function chart implementation should allow the division of the code into smaller parts. For example, each machine of a complex line to be controlled may have its own graph, and the graphs for several machines could then be assembled together. Such hierarchical structuring is of

fundamental importance when programming the operation of large, complex systems.

Function charts are not only suitable for complex operations, but can also be very useful for simpler tasks. A function chart is quite easy for the non-specialist to understand. An accepted standard for the description of automated operations also has the advantage that more computer code can be maintained and re-utilized and does not need to be written anew each time, as would be the case with incompatible devices and languages.

The translation of function charts to computer code depends on the specific PLC and its tools, as not all devices have such compilers. Still, even if the function charts cannot be transformed in programming code, the diagrams are very useful, since they provide the user with a tool to analyze and structure the problem. Some companies use function charts to describe the function and use of their equipment. Of course, it would be much simpler if function charts would be used all the way from functional description to actual programming.

5 Combinatorial Circuits

Switching theory provides a model for the operations of binary elements, i.e., those that can be only in one of two possible states. The books [7] and [8] are well-known texts on the subject. There are several examples of binary components. Binary circuit components like switches, relays, and two-position valves to be used in logic circuits are designed to operate in two states only. A transistor is basically not a binary component, but it can be operated as a binary element, if only the states "conducting" and "not conducting" are considered.

The state of a binary element is indicated by a binary variable that can consequently only take two values, conventionally indicated as "0" or "1". For a switch contact, relay contact, or a transistor (represented by a Boolean variable x) the statement $x = 0$ means that the element is open (does not conduct current) and $x = 1$ means that the element is closed (it conducts a current). For a push button or a limit switch, $x = 0$ means that the switch is not being actuated, while $x = 1$ indicates actuation. A binary variable can also correspond to a voltage level in a practical circuit implementation. In "positive logic" the higher voltage level corresponds to a logical "1" and the lower level to a logical "0". In transistor-transistor logic (TTL), binary "0" is usually defined by a voltage level between 0 and 0.8 V and binary "1" by any voltage higher than 2 V. Similarly, in pneumatic systems $x = 0$ may mean that a line is at atmospheric pressure and $x = 1$ that the line is at higher pressure.

Standardized symbols are used to represent logic (combinatorial and/or sequential) circuits independently of the practical implementation (with electric or pneumatic components). This type of representation is called a *function*

block. There are international standards for the logic symbols, IEC 113-7 and IEC 617; many other national standards are also defined on their basis.

An SFC is used to realize a sequence of operations, where each step corresponds to a finite state of the system. The control (or object) part of the SFC often consists of a Boolean expression or a logical circuit. Such an expression can be expressed as a combinatorial circuit. It consists of several logical expressions, in which the output value y depends only on the current combination of the input signals $u = (u_1, u_2, \ldots)$:

$$y(t) = f[u(t)].$$

Note that this is an algebraic condition and there are no states defined. The number of Boolean functions grows rapidly with the number of variables n, since the number of combinations becomes 2^n. There are different methods for the simplification of Boolean functions, in which the number of the variable relations is reduced. It is outside the scope of this text to discuss in detail the different simplification methods for Boolean functions, but references are given at the end of the text.

5.1 Representation using LDs

The implementation of Boolean expressions can be programmed in LD, which now make up a part of the international standard IEC 61131-3. An LD consists of graphic symbols, representing logic expressions, and contacts and coils, representing outputs.

Relay circuits are usually drawn in the form of wiring diagrams that show the power source and the physical arrangement of the various components of the circuit (switches, relays, motors, etc.) as well as their interconnections. The wiring diagrams are used by technicians to do the actual wiring of a control panel. The LD is a widely used representation form for logical circuits. It represents a conventional wiring diagram in schematic form, without showing each electrical connection explicitly. In an LD each branch of the control circuit is shown on separate horizontal rows (the "rungs" of the "ladder"), as shown in Fig. 4.

Each branch reflects one particular function and the related sequence of operations. In this drawing frame it is implicitly assumed that one of the vertical lines is connected to a voltage source and the other to ground. Note that all the rungs of the ladder are "executed" simultaneously in a wiring diagram.

In the LD are shown relay contacts that can be either of *normally open* or *normally closed* type (the normal state is the one in which the coil is not energized). The output consists of a relay (a coil) that could also symbolize a more complex circuit or a flip-flop. The drawing symbols for the switches and an actuator (relay) are shown in Fig. 5.

Example (A combinatorial circuit) An LD can represent a combinatorial circuit (see Fig. 6). The series connection of the switches represents a

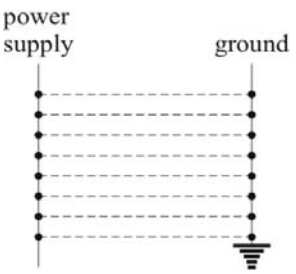


Fig. 4. The framework of a ladder diagram (LD). This is the way that the wiring of relay schemes was built up. The same structure is implemented into PLCs but with logical instructions.

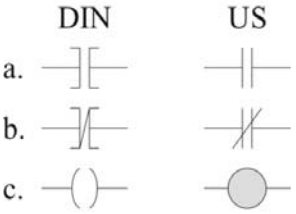


Fig. 5. The German DIN and the U.S. standards for symbols for (a) a normally open contact, (b) a normally closed contact, and (c) an output element, such as a relay coil, in an LD.

logical AND and the parallel connection a logical OR. The variables u_1, u_2, and u_3 indicate the input contacts and Y_1, Y_2, and Y_3 the output relays. The variables y_2 and y_3 denote the corresponding logical variables, stored in the computer. All the input conditions, i.e., the activation of the switches, must be satisfied simultaneously.

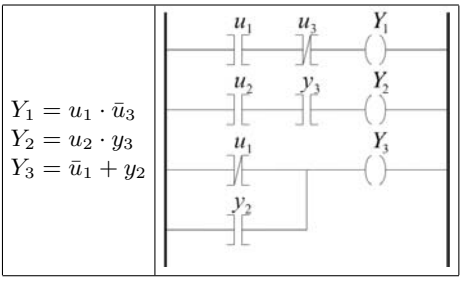


Fig. 6. A combinatorial circuit represented as an LD. The power supply and the ground are now symbolized by the vertical lines. The LD is interpreted as Boolean expressions. The notation "·" indicates a logical AND while "+" denotes OR.

The input switch contacts usually have negligible resistance and can be, e.g., pushbuttons, limit switches, or pressure or temperature sensors. The output element could be any resistive load (e.g., a relay coil) or a lamp, motor, or any other device that can be electrically actuated. Each "rung" of the LD must contain at least one output element; otherwise, a short circuit between power supply and ground will take place.

6 Basic Structure of PLCs

The basic operations of a PLC correspond to the combinatorial control of a logical circuit. In addition, a modern PLC can carry out other operations such as counting, the processing of signal delays, and a wait for defined time intervals. The major advantage of a PLC is that a single circuit with its compact construction can replace hundreds of relays. Of course, another advantage is that the PLC is programmable and not hardwired, so that its operation can be changed with limited effort. PLCs can, on the other hand, be slower than hard-wired relay logic. An optimal solution for each specific application can be realized when both technologies are installed in the same system, so that the advantages of each can come to use.

The PLC hardware is usually built to fit a typical industrial environment, especially regarding signal levels, heat, humidity, unreliable power supplies and mechanical shocks and vibrations. PLCs also contain particular interfaces for conditioning and preprocessing of different signal types and levels. PLC functionality is also being increasingly offered in process input/output units, which are connected to larger integrated control systems.

6.1 PLC instruction list

A PLC must operate in real time. The input and processing of external signals can take place in two ways in a PLC: by polling (repeated requests) or via interrupt signals. Polling has the drawback that some external event(s) may be missed if the PLC is not sufficiently fast. On the other hand, such a system is simple to program. An interrupt-driven system is more difficult to program, but with this system the risk of missing some external event is much smaller. In simpler automation systems polling is usually more than adequate; whereas, interrupt-driven control is used in more complex control situations.

The programming of a PLC consists mainly of defining control sequences. The input and output functions are already implemented in the PLC basic software. The assembler-like instructions are translated in the PLC-to-machine code. At execution time the program is run cyclically in an infinite loop. In this way it is simulating the parallelism inherent in the wired relay logic. The *read-execute-write* cycle is called a scan cycle. Every full scan may take about 15–30 ms in a small PLC; this time is approximately proportional to the program size in memory.

The response time of the PLC depends on the time that is necessary for processing the program code. During the scan cycle the PLC processor cannot read any new input signals or output new control signals. Given that the scan cycle time is short enough with respect to the time constants of the plant, this effectively simulates the parallel behavior in the hardwired relay logic that the PLC was meant to replace. For an outside viewer, and specifically the controlled process, all outputs seem to change their values simultaneously in response to the input signals.

The ladder rungs are evaluated in a fixed order, beginning with the first rung. When the scan cycle has finished, the intermediary output-memory region is then copied to the physical outputs by the PLC hardware in one operation. It is important to note that the PLC evaluates the rungs sequentially, usually from top to bottom and from left to right. This means that previously evaluated rungs can affect the evaluation of the current rung, even though the results of those previous rungs are not yet shown to the outside process. As a consequence, it is usually a mistake to assign the same output from two different rungs. Some PLCs warn about such programming mistakes.

A small number of basic machine instructions can solve most sequencing problems. A program that contains these instructions is called an *instruction list (IL)*. A PLC has pins that are assigned to inputs and outputs, connected to the physical process. The input signals are first read into a buffer memory register. This function is always included in the PLC system software and does not need to be explicitly programmed by the user. Some of the fundamental instructions are listed here; usually they can operate on bits as well as on bytes.

- **ld, ldi**: Loading of a value from one input buffer memory into the accumulator, direct (ld) or inverted (ldi);
- **and, ani**: AND or inverted AND instruction between the value in the accumulator and the value of an input channel; the result is stored in the accumulator;
- **or, ori**: An OR or inverted OR instruction between the value in the accumulator and the value of an input channel; the result is stored in the accumulator;
- **out:** The content of the accumulator is copied to the output buffer memory and controls the output signals. The value is also retained in the accumulator so that it can be further processed or sent to other output ports.

7 PLC Programming with Ladder Diagram and Instruction List

The logical control instructions for the PLC can be expressed in the form of an LD as well as with an IL. The gate y_1 is used to give memory capability to the relay Y_1 (self-holding capability). See Fig. 7.

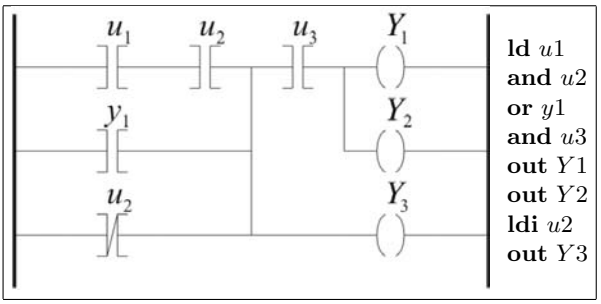


Fig. 7. PLC control instructions in the form of an LD and as an IL.

8 Sequencing Circuits Describing Ladder Diagrams

In the previous sections a sequential function chart was used to implement states and transitions of a sequencing circuit. They can also be programmed in a low level language, the LD. As a consequence we need a way to represent a state within an LD. The building block for the sequencing control is the set-reset (SR) flip-flop circuit.

A flip-flop can be described by an LD (see Fig. 8). When a set signal S is given (i.e., a set switch is pressed), the S switch conducts a current that reaches the relay coil Y; the input R is so far not activated. The energization of the relay coil leads to the closure of the relay contact y in the second rung. If the S switch is now released, a current still continues to flow to coil Y via the contact y and the flip-flop remains set. The y contact acts as the "memory" of the flip-flop. By pressing the reset switch R, the circuit to the coil Y is broken and the flip-flop returns to its former reset state. In industrial practice such a relay is called bistable, self-holding, or latched.

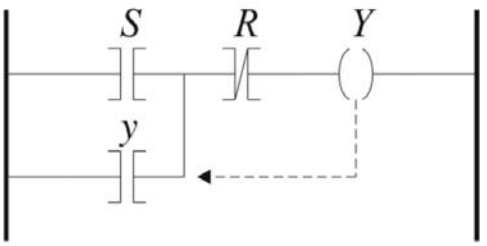


Fig. 8. An SR flip-flop, represented in the form of an LD (self-holding or latched relay)

Since only one state at a time can be active, some kind of execution control signal is necessary in order to change from one state to another. In other words, a transition has to be realized. This type of control signal can be given when a condition is satisfied (the condition could of course also be a complex combination of control signals). The conditional order acts at the same time as a *reset* (R) signal for one step and also as a *set* (S) signal for the following step. The sequencing control execution can therefore be described as a series of SR flip-flops, where each step corresponds to a rung of the ladder, as shown in Fig. 9. At each execution control signal, the next flip-flop is set. The execution proceeds one step at a time and after the last step returns to the beginning of the sequence (step 1).

In practical execution, step 1 is initiated with a *start* button or, in the case of a closed execution loop, automatically after execution of the last step. When the last step is active and the condition for the jump to the first step is satisfied, then the step 1 coil is activated, and the self-holding relay also keeps it set after the first condition no longer holds. The output of the first step also activates the input contact "step 1" that is connected in series with the contact for the condition for step 2. As soon as this condition is satisfied, the relay step 2 latches circuit 2 and at the same time opens the circuit for step 1. The following steps are carried out in the same fashion. Obviously, in order to ensure a repetitive sequence, the last step has to be connected to step 1 again.

This type of execution is called *asynchronous*. In switching theory a synchronous execution is also considered, in which the state changes are controlled by a time clock. An asynchronous system is thus known as *event* based, while a synchronous system is *time* based. In manufacturing automation applications, asynchronous control is much more common, since the operation of most machines and equipment (and thus their state changes) depends on a set of conditions rather than on a stiff time plan. In the design of control sequences it is also important to consider that the conditional input signals must keep their logical level for the full duration of the corresponding operation. If this is not the case, then buffering or intermediate storage must be provided for the input signals.

9 PLC Programming

PLCs can be programmed in different ways: with the assembler-like IL or in higher, problem-oriented Structured Text (ST). We have demonstrated that both combinatorial networks and sequences can be described using LDs. The LD has been particularly popular in the U.S.A., while in Europe the use of function block diagrams (FBDs) with the graphical symbols for logical gates is more common. The high level description of sequencing functions

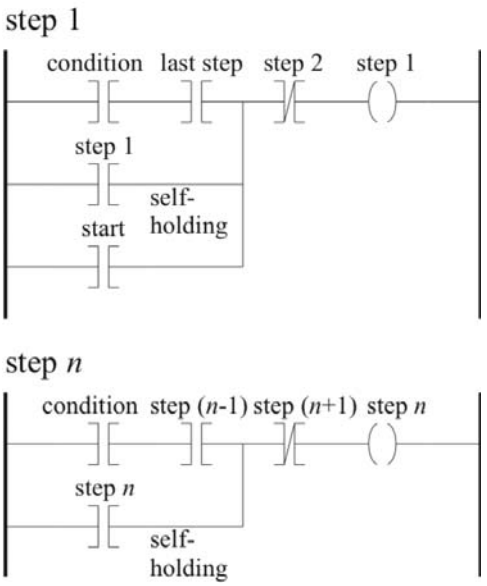


Fig. 9. Sequencing execution described in LD notation

using the Grafcet-like sequential functions chart (SFC) is naturally gaining in popularity.

PLCs are usually programmed via external units. These units as a rule are not needed for the PLC on-line operation and may be removed when the PLC is in operation. Programming units are typically small, hand-held portable units or portable personal computers. A manual PLC programmer looks like a large pocket calculator with a certain number of keys and a simple display. Each logic element of the LD or program instruction is entered with specific keys or key combinations. More sophisticated programming can be performed with a PC, offering both graphical and text editors.

The international standard IEC 61131-3 (earlier called IEC 65A (SEC) 67) is the only global standard for industrial control programming. It harmonizes the way people design and operate industrial controls by standardizing the programming interface. A standard programming interface allows people with different backgrounds and skills to create different elements of a program during different stages of the software lifecycle: specification, design, implementation, testing, installation, and maintenance. Yet all pieces adhere to a common structure and work together harmoniously.

IEC 61131-3 includes the definition of the SFC language, used to structure the internal organization of a program, and four interoperable programming languages: IL, LD, FBD, and ST. ST has a formal syntax similar to that of the programming language Pascal, as shown in this short example:

IF TEMP1 > 50.0 THEN

```
Flow_ rate := 65.0 + OFFSET
ELSE
Flow_rate :=75.0; PUMP:= ON;
END_IF;
```

ST supports a wide range of standard functions and operators. ST and IL represent algorithmic formulations in clear text. The FBD, the LD, and the SFC are instead graphical representations of the function and the structure of logical circuits. The international standard IEC 61131-3 should therefore guarantee a wide application spectrum for PLC programming.

By using a graphical editor an SFC can be programmed readily. The actions can then be programmed in some other 61131 language definitions, such as an ST or LD piece of code. FBD is also a popular complement to the SFC. The actions can also consist of special regulator algorithms, such as a PID controller.

PLCopen

PLCopen is a vendor- and product-independent worldwide association. Its mission is to be the leading association resolving topics related to control programming to support the use of international standards in this field. To achieve this, PLCopen has several technical and promotional committees. PLCopen was founded in 1992 [12]. One of the core activities of PLCopen is focused on IEC 61131-3.

SoftPLC

In 1997 several automation software producers started to market products called SoftPLC [13]. The partners are called authorized SoftPLC integrators and they offer SoftPLC integration services. A number of vendors supply SoftPLC compatible products. The basic idea is to make use of the enormous development of the PC hardware. The cost of a common PC is very low today and gives a remarkable computing power for the price. Also, the PC hardware of today has quite a high quality. There are many reasons to take advantage of this development for the benefit of automation. The PLC market is much smaller, and the PLC development has a hard time competing with the PC market.

At the same time, distributed I/O has become increasingly common. Distributed I/O and "intelligent" sensors and actuators make is possible to replace the traditional signal wire with a digital information carrier. If the PC controls its I/O units via a fieldbus, there is no need for more measurement and actuator cables between the PC and the physical process.

The PC is not primarily designed for real-time applications. However, due to its high performance, it can still be used in many demanding automation applications. The hardware has a large computing capacity and advanced graphics capabilities are available. This makes it possible to program, debug,

and document the software according to the standard IEC 61131-3. The powerful hardware gives the SoftPLC implementation quite a high performance. It is relatively straightforward to couple the control system to systems for material requirements and planning and to graphical operator interfaces since there are data base systems already available for PC platforms.

10 Summary

The PLC development has been remarkable—from simple machines performing only binary operations to a wide spectrum of PLC systems. There are now units for simple and basic operations as well as complex automation computers.

The software development has now reached a maturity that is demonstrated in the international standard of IEC 61131-3. A PLC can be programmed in a structured way, allowing a mixture of both sequential function charts that offer a program structure and four other low and high level languages that permit both algorithmic and graphical representations of the actions to take place. Furthermore, complex sequencing and parallel processes can be readily programmed, and hierarchical representations are possible.

References

1. R. W. Lewis. *Programming Industrial Control Systems Using IEC* 61131-3. IEE, Stevenage, Herts, SG1 2AY, UK, 1998.
2. K. H. John and M. Tiegelkamp, M. *PLC Programming with IEC* 61131-3: *Concepts and Programming Languages, Requirements for Programming Systems, Decision-Making Tools* Springer-Verlag, Berlin and Heidelberg, 2001.
3. F. Petruzella. *Programmable Logic Controllers,* 2nd Ed.*,* McGraw-Hill Publishing Co, New York, 1998.
4. W. Bolton. *Programmable Logic Controllers*, 3rd Ed., Newnes, Oxford, 2003.
5. T. A. Hughes. *Programmable Controllers*, 3rd Ed., Instrument Soc. of America, Research Triangle Park, NC, 2000.
6. *Programmable Controllers-Part 3: Programming Languages,* 2nd Ed., International Electrotechnical Commission (IEC), 2003.
7. W. I. Fletcher. *An Engineering Approach to Digital Design*, Prentice-Hall, Englewood Cliffs, NJ, 1980.
8. Z. Kohavi. *Switching and Finite Automata Theory,* 2nd Ed., McGraw-Hill, Inc., New York, 1978.
9. A. S. Tannenbaum. *Computer Networks,* 3rd Ed., Prentice-Hall, Englewood Cliffs, NJ, 1996.
10. J. Bentham. *TCP/IP Lean: Web Servers for Embedded Systems,* 2nd Ed., CMP Books, 2002.
11. J. Berge. *Fieldbuses for Process Control: Engineering, Operation and Maintenance*, Instrument Soc. of America, Research Triangle Park, NC, 2001.
12. PLCopen home page www.plcopen.org.
13. SoftPLC home page www.softplc.com.

Digital Signal Processors

Rainer Leupers[1] and Gerd Ascheid[2]

[1] RWTH Aachen, Software for Systems on Silicon
[2] RWTH Aachen, Integrated Signal Processing Systems
{leupers,ascheid}@iss.rwth-aachen.de

1 DSP Applications

As the name digital signal processor (DSP) suggests, the main application area is digital signal processing. The need for such processing occurs in a variety of domains including:

- Wireless communication
- Wireline communication
- Video and imaging
- Audio
- Security (e.g., biometrics)
- Digital control
- Automotive
- Measurement/sensing.

More important than an exhaustive list of applications are the characteristics of digital signal processing to understand the specifics of DSPs. Typically, *algorithms* are applied to a signal, e.g., to extract information, to get a different representation, and to "clean" or to shape the signal. Examples of frequently used algorithms are time variant and time invariant filtering, transformations like the fast Fourier transform (FFT) or discrete cosine transform (DCT) or parameter estimation. A special type occurring in a variety of algorithms is the vector scalar product, which combines multiplication and accumulation (MAC):

$$\sum_{k=0}^{N-1} a(k)b(k) \quad \text{or} \quad \sum_{k=0}^{n} a(k)b(n-k).$$

In general, signal processing algorithms require execution of mathematical operations, e.g., multiplications, additions, and nonlinear functions like sine, cosine, ln or exp on real and complex numbers and, for example in coding, may

even include finite field arithmetic. Usually, these operations have to be performed on larger sets of data (the signal). While in some applications off-line processing may be possible, most digital signal processing is *real-time processing* leading to minimum throughput and maximum latency requirements.

Over time, algorithms have become more and more sophisticated to get as close as possible to the (theoretical) optimum of the processing result. This often means that algorithms are adapted on-line to the detected scenario. In multi-standard and multi-application devices, various signal processing algorithms are used. The required flexibility is a strength of DSPs.

Basically, DSPs can cover a wide range of throughput requirements. High end DSPs are able to execute several billion instructions per second due to a combination of high clock rate and a high degree of parallel processing. But high throughput has its price: high power consumption and high parts cost. Since many DSP applications are battery-operated mobile devices (e.g., cell phones, WLAN) and high volume products, this is an issue. Therefore, the trade-off is always between price and power consumption on one hand and processing speed on the other. The cost of high throughput can be reduced significantly if a more specific solution is used. The range is from a *dedicated hardware implementation* (least programmable, minimum power consumption), via an *application specific processor* (optimized for an application), to a *general-purpose DSP* (most flexible, maximum power consumption).

2 DSP Architectures

2.1 Basic considerations

DSP architectures have developed significantly over time. With the progress of silicon technology, it became possible to implement more sophisticated features and functionalities and to use more parallel processing to increase the throughput. Most DSPs, however, are based on an architecture and components as shown in Fig. 1. This architecture addresses the specific requirements of digital signal processing.

Bus architecture:
Since large amounts of data have to be processed and throughput is critical, DSPs have separate program and data buses. Mathematical operations often require two or more operands. Therefore, DSPs have multiple data buses. A typical number is 3 to support reading of two operands and writing of a result in the same cycle.

Algorithmic units
The basic operations usually supported by dedicated hardware components are logical operations, addition/subtraction, and multiply-and-accumulate (see Section 1) as well as some manipulations of particular importance for integer arithmetic like shifting, rounding, and truncation.

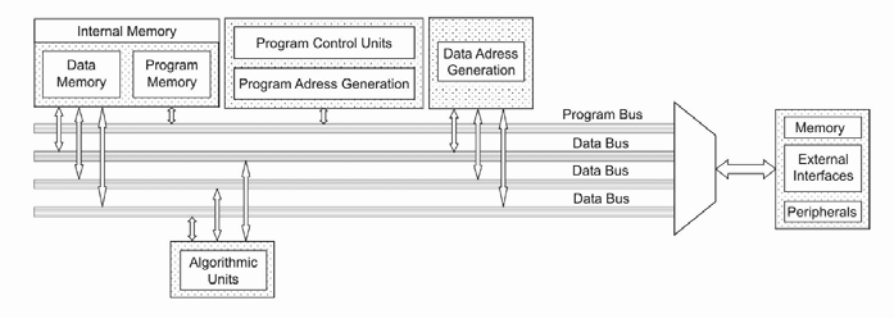


Fig. 1. Basic DSP architecture

Sometimes a separate multiplier (in addition to the multiplier in the MAC unit) is available.

Program address generation

Besides the common functions for program address generation, DSPs usually support zero-overhead loops. As was stated earlier, signal processing often is done on large sets of data. Therefore, loops, where the same processing is repeated multiple times, are common. To speed up execution, loop control—decrementing the loop counter and conditional branching—are taken care of by the program address generation unit. Thus, these operations do not consume extra cycles (except for the initialization) or arithmetic unit resources.

Data address generation

In a DSP the data address generation unit supports several schemes typical for signal processing. The general pattern for processing sets of data is that the data address is stored in a register and, after data access, the data address generation unit calculates the next data address. To support this, the data address generation unit provides a number of addressing registers and address arithmetic units: increment/decrement, addition/subtraction, and modulo arithmetic. Since the address arithmetic is done by hardware blocks in the address generation unit, neither additional cycles nor arithmetic unit resources are consumed for address generation, which is important to achieve high throughput.

2.2 Pipelining

A key concept to increase the throughput is pipelining [1], which is standard in DSPs. To execute an instruction:

1. The instruction word is fetched from the program memory.
2. The instruction word is decoded.
3. Operands (if any are required by the instruction) are fetched.
4. The instruction is executed.

5. The result of the operation is written back.

Instruction and operand fetch require at least one clock cycle each. Therefore, significant speed-up is achieved by breaking down the instruction into a number of pipelined steps, as shown exemplarily in Fig. 2. While an instruction is decoded, the next instruction is already fetched. Once the pipeline is filled, one instruction per cycle is completed.

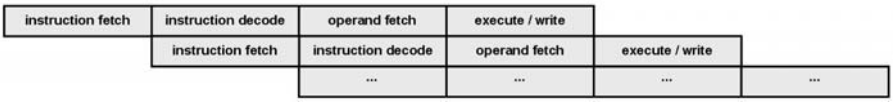


Fig. 2. Instruction pipelining

TI's TMS320C54x series, for instance, uses six pipeline stages (similar to the partitioning in Fig. 2 but with two pipeline stages instead of one for each of the two fetch steps). Note that with pipelining three data buses are advantageous: a two-operand fetch of an instruction in the operand fetch stage may occur together with a write from an instruction in the final pipeline stage.

Unfortunately, pipelining can overturn the order of execution and, thus, cause wrong results if not handled properly. For example, if an instruction needs operands written by the immediately preceding instruction, old operand values will be fetched (data hazard) since the preceding instruction is still in the execute/write stage at that time (see Fig. 2). Similarly, instructions following a branching instruction enter the pipeline before the branch instruction is decoded and the program control unit switches to the appropriate instruction (control hazard). There are two ways to resolve such hazards: pipeline control by the programmer (or a compiler) or by the DSP hardware (interlocking). TI, for example, uses hardware pipeline control.

With growing processor speed and increasing memory sizes, memory access becomes the limiting factor for the clock speed. To overcome this bottleneck, advanced DSPs use one or even two levels of caching. Detailed descriptions of cache design are found, for instance, in [1].

2.3 Arithmetic

DSPs are available with floating-point arithmetic as well as with fixed-point arithmetic. Recalling the basic floating-point number format (Fig. 3), it is obvious that the arithmetic is more complex (e.g., adjustment of operands before operations and re-normalization of results after operation). Therefore, floating-point arithmetic is typically slower and consumes more power. Floating-point DSPs usually also support fixed-point arithmetic. However, the floating-point operations use more pipeline stages than the fixed-point operations to achieve the same clock rate.

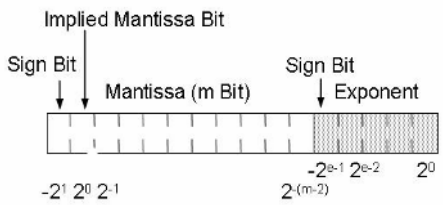


Fig. 3. Floating-point number representation

Especially with fixed-point arithmetic, the range expansion of operations needs to be taken care of. A 16x16 bit multiplication, for instance, yields a 32 bit result. Before storage the result has to be rescaled to the original size. The algorithmic unit of a DSP supports different types of rescaling, e.g., rounding, truncation, or saturation. For operations like multiply-and-accumulate it is preferable to perform the rescaling only after the final accumulation. This is made possible by providing accumulation registers with more than twice the word length, e.g., 40 bit registers in a 16 bit DSP.

In comparison, software development is faster for floating-point arithmetic, since for fixed-point arithmetic, additionally the parameters (constants) have to be converted to fixed point, and scaling of operands as well as truncation/rounding have to be specified without degrading the (algorithm) performance unacceptably. The trade-off between floating-point and fixed-point DSPs, therefore, is between faster code development and, sometimes, better algorithm performance for floating-point DSPs, and lower cost per part and lower power consumption for fixed-point DSPs.

Since signal processing involves arithmetic on large sets of data, throughput can be significantly increased by providing multiple arithmetic units which can execute in parallel or by including application-specific arithmetical functions. This will be revisited when discussing advanced architectures.

2.4 Data address generation unit

The general concept of the data address generation unit was already discussed in the introductory Section 1. Two very DSP-specific functions should be considered in more detail.

Algorithms like the MAC equation shown in Section 1 are often calculated on an ongoing basis. Since we cannot store permanently additional new values in the memory, only the signal values needed for one step are stored. After each step, the new signal value needed for the next step is stored such that it replaces the oldest value, which is not required in the next step anymore. To have a consecutive storage of data with the the newest signal value always at the lowest (or highest) memory address of the range, the complete data set would have to be moved each time a new value is stored. In such a case it is more efficient to skip the data shift and create a “circular buffer.” Without

shifting the data, the pointer to the newest value moves through the data set and, after reaching the end address of the data set range, continues at the beginning of the data set range. The consecutive ordering of the data then is also from the position of the newest value to the end of the data set range and continuing at the beginning of the data set range. This means for the address generation that the offset from the data set start address must be incremented using a modulo-addition:

```
Offset(k+1) =  {Offset(k) + Increment} modulo Bufferlength
```

Since this is a frequent operation in signal processing applications, DSPs usually provide the required hardware in the address generation unit. Thus, the circular buffer arithmetic does not require additional cycles or arithmetic unit resources for execution.

A second DSP-specific address generation function is bit reversed addressing. It is a particular way of bit permutation in the addresses between data accesses for an efficient implementation of an FFT. Again, to support fast execution, many DSPs provide this function as part of the address generation unit.

2.5 General-purpose DSP architectures

In general-purpose computing, reduced instruction set computing (RISC) architectures are dominating. The name RISC stems from the original idea to reduce the number of instructions by providing only frequently used instructions. This approach makes it possible to build faster CPUs. Although the missing instructions now have to be emulated by several instructions, a significant overall throughput gain remains. Today the key characteristic of RISC architectures is that all operations on data exclusively apply to data in registers. Loading data to registers and storing data from registers is done by separate load/store operations. Thus, load/store and data manipulation operations are separated (for this reason, RISC is also referred to as "load/store" architecture).

Since signal processing often applies the same operation to large sets of data, a load/store architecture offers fewer advantages for such applications. Therefore, most DSPs do not fall into this category. Instead, advanced DSPs employ parallelism to increase throughput. Besides a parallel utilization of available resources, speed-up is achieved by providing resources multiple times, e.g., several multipliers, arithmetic logic units (ALUs) or MAC units in parallel. Instructions then must support parallel execution.

With *single instruction multiple data* (SIMD) architectures, one instruction triggers parallel execution of the same operation on multiple data. Since the parallel execution increases the throughput, the load/store throughput must also be increased by either increasing the bus width or adding separate data buses.

On this basic level, SIMD is limited in its efficiency. Taking the specifics of a particular application into account, the SIMD concept can be employed more effectively (this will be discussed in the next section). In general, operation on multiple data in parallel should be combined with instruction level parallelism, whose two basic flavors are superscalar and very long instruction word (VLIW) processors.

Superscalar processors read blocks of instructions and control the distribution of the instructions to the parallel processing units and the avoidance of hazards by hardware at run time. The instructions do not differ from the instruction of a scalar processor.

VLIW processors offer long instruction words which are either groups of a fixed number of instructions or fixed instruction packets with parallelism explicitly indicated by the instruction. Thus, either the programmer or the compiler has to take care of parallelism and hazards.

While for advanced general-purpose processors the superscalar architecture is favored, advanced DSPs prefer the VLIW concept and its variants (e.g., using compressed long instruction words, see, e.g., [6]). More information about the architectures of commercially available DSPs is found in the user manuals, which, in general, are available via internet, e.g., [3], [4], [5]. For an overview of DSP features, see, e.g., [2].

2.6 Application-specific processors

When the design of an ASIC for a particular product is out of scope, e.g., for cost reasons, a stand-alone DSP possibly combined with a field programmable gate array (FPGA) for hard-wired functions is the solution of choice. A general-purpose DSP, however, is not the most efficient solution in terms of cost and power consumption. Therefore, some vendors are addressing particular application domains by adding specific functions to the DSP. An example of such a function is the add compare select required in Viterbi decoding available in several TI DSPs.

Where part cost (in particular, in high volume products) and power efficiency (e.g., in battery-operated devices) are critical, an application-specific design is the only choice. In these cases the developed IC is a system-on-chip, as shown exemplarily in Fig. 4. A μ-controller for user application software and control is combined with a DSP, hardware accelerators, and specific interface blocks. The signal processing is either mapped on the general-purpose DSP functions or on specific hardwired blocks. Here we have a number of options: a separate hardwired block, an application-specific co-processor, or the option of tailoring the instructions of the processor to the application, i.e., to implement an application-specific instruction set processor (ASIP). Shorter development time for increasingly complex products, on-going revisions and enhancements of standards (like, for example, wireless communication standards), and multi-standard devices can only be handled with the flexibility of

programmable devices. On the other hand, throughput requirements must be met at an acceptable power consumption level.

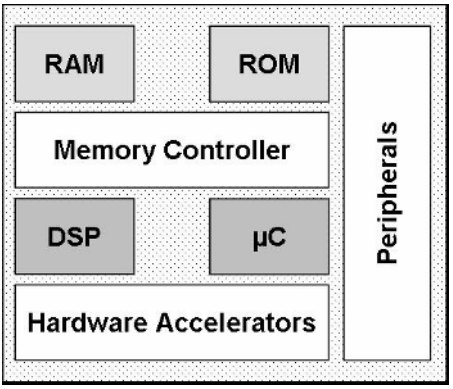


Fig. 4. DSP-based application-specific SoC

A hardwired implementation typically yields the best power efficiency but has the least flexibility. A general-purpose DSP offers the highest flexibility but at high power consumption. An ASIP (and an application-specific co-processor can also be seen as an ASIP) is an excellent compromise between DSP and hardwired implementation in terms of flexibility and offers significantly better power efficiency (MIPS/watt) over a general-purpose DSP.
Examples of different ASIP architectures are:

Specific hardware resources, e.g., an application-specific algorithmic unit. Examples are the add-compare-select function for Viterbi decoders, the CORDIC for rotation of a complex phasor, or finite field operations for block codes.

Complex processing blocks with separate execution control. Processing in such a block is initiated by a specific instruction and may finish in a known number of cycles or, for example, indicate completion by a flag or an interrupt.

Exploration of application-specific parallelism. In particular for high throughput applications, algorithmic units should be implemented multiple times to exploit the inherent parallelism of the signal processing algorithms. Examples are parallel calculation of sections of an FFT, parallel filtering of parallel signal paths, etc. To avoid the need for dynamic scheduling of instruction execution (and, thus, additional hardware), the most suitable instruction set architecture for these ASIPs is VLIW/SIMD.

3 DSP Architecture Design

3.1 Application-specific architecture design methodology

While DSP applications could be implemented on any processor architecture, designing an application-specific architecture requires a special methodology and tools. Only in this way can one achieve an optimal hardware/software solution that meets all design constraints while minimizing costs. In fact, the design of application-specific DSPs can be viewed as a problem of *hardware/software co-design* at the processor architectural level.

While some standard recipes for DSP processor design result from common properties of the application domain (e.g., multiply-accumulate instructions for sum-of-products computation or cyclic memory addressing for filter computation), determining the detailed processor features requires a more thorough examination of the target application. Therefore, today's state-of-the-art methodology is *architecture exploration*. As illustrated in Fig. 5, this denotes an iterative, profiling-based design flow during which a *successive refinement* of the architecture is performed.

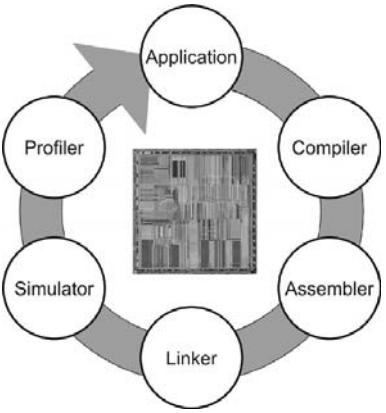


Fig. 5. Processor architecture exploration

The initial architecture may be a legacy processor, a customizable RISC core (such as, e.g., Tensilica's Xtensa), or an "educated guess" based on the designer's experience. The most reliable way of evaluating a processor architecture before fabrication is to simulate the execution of the application with a *virtual prototype* of the architecture running on a host machine. This requires that the application be mapped to executable code via the usual software development tool chain (see Section 4) including compiler, assembler, and linker. An instruction set simulator (ISS) with profiling support permits one to identify *execution bottlenecks* and promising *application-specific machine*

instructions. The designer can accordingly modify the virtual prototype and iterate the exploration loop until an optimal architecture has been determined.

While the architecture exploration methodology yields highly optimized results, its potential bottleneck is the need to adapt the software development tools, including the ISS and profiler, in each iteration to a modified target architecture. This *retargeting* usually requires significant manpower, since the consistency and correctness of all tools involved must be ensured before the next loop iteration can take place.

3.2 Architecture modeling and design tools

The need for fast software tool retargeting in architecture exploration has motivated a special class of *electronic design automation* (EDA) tools that support application-specific processor design. As an example, we show the design flow of the LISATek Processor designer (CoWare) in Fig. 6.

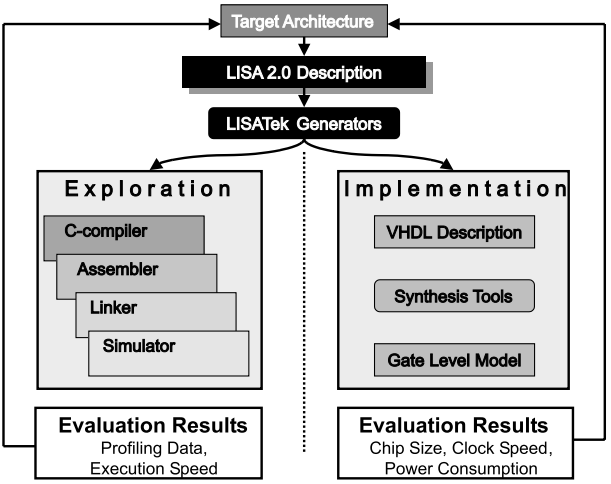


Fig. 6. LISATek processor design flow

The key idea is to use an *architecture description language* (ADL) to model the target architecture and its instruction set at a high abstraction level. In the case of LISATek, this is LISA 2.0, a C/C++ based ADL. A LISA 2.0 model comprises *processor resources* (including registers, memories, and instruction pipelines) as well as *operations* that describe the transitions of the processor state when executing instructions. Among others, the operations capture the binary coding, assembly syntax, and abstract behavior of instructions. Fig. 7 shows an example of a LISA 2.0 operation model for a logical shift left (SLL) instruction.

```
OPERATION SLL IN pipe.EX {
  DECLARE
    {
      GROUP rs1, rs2, rd = { reg32 || reg16};
    }
   CODING { 0b1001 rs1 rs2 rd 0bx[5] 0b110010 }
   SYNTAX { "SLL" rd "," rs1 "," rs2 }
   BEHAVIOR
    {
              temp=(unsigned int)rs1;
              rd = temp << ( rs2 & 0x0000001f);
    }
}
```

Fig. 7. Excerpt from a LISA 2.0 ADL model

After modeling the target architecture in LISA 2.0, the LISATek generators can automatically retarget all software development tools. Due to the use of a single reference ADL processor model, inconsistencies are avoided. In addition, architecture exploration (left loop in Fig. 6) can be performed efficiently. In order to enable a path to hardware implementation of the target processor, LISATek can also generate register-transfer level (RTL) processor models in *hardware description languages* (HDLs), such as VHDL, Verilog, or SystemC. Based on the generated RTL HDL model, a usual VLSI design flow, including gate-level synthesis and optimization, can be used to finalize the implementation. The right loop in Fig. 6 indicates that results from hardware synthesis, like cost and performance metrics, can be back-annotated into the exploration phase. This is required, since accurate metrics are available only after synthesis, and the architecture can be fine-tuned according to these metrics.

The software development tool environment generated by LISATek is also used for application programming on the final processor architecture, either in assembly or C (see Section 4). Fig. 8 shows a typical setup of the graphical environment during application code simulation and debugging. It displays C source code and assembly code, as well as the processor state (memory and register contents).

4 DSP Programming

4.1 Assembly programming

For efficiency, programming at the machine instruction level is still frequently used for DSPs. *Assembly language* is a symbolic machine-level programming language, where code consists of human-readable acronyms (*mnemonics*) like ADD, SUB, LOAD, etc. Each mnemonic represents a single instruction of the target machine.

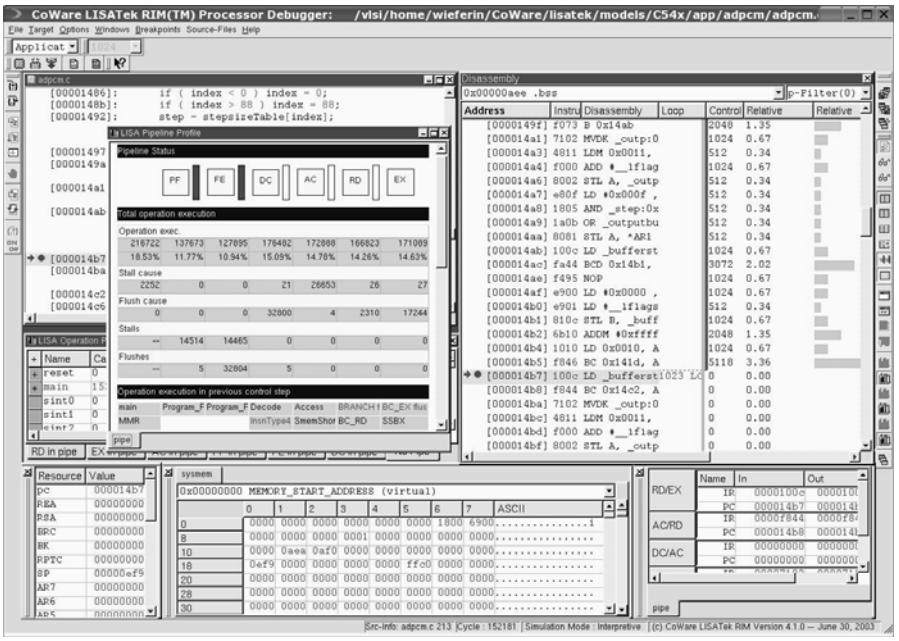


Fig. 8. LISATek Processor Debugger

The translation from symbolic assembly code to binary machine code is done by an *assembler*. Besides performing a one-to-one mapping from mnemonics to binary operation codes (*opcodes*), the assembler is also responsible for computing addresses of symbolic labels and reserving memory segments for data and variables. In order to distinguish user commands, e.g., for memory reservation, from valid assembly instructions, the assembler makes use of *pseudo-instructions* or *directives*. These are mostly indicated by a leading dot. For instance, ".ORG" is usually used to specify the start address ("origin") of a program in memory. The output of the assembler is *object code*. Different object code modules can be assembled independently and are finally glued together (possibly also using subroutines stored in libraries) by the *linker*, which emits executable binary programs.

What makes the use of assembly programming for DSPs different from other microprocessor families (e.g., RISC) is the fact that DSP machine instructions are usually very specific to the application area. Therefore, a large "semantic gap" exists between general-purpose high-level programming languages like C (see Section 4.2) and DSP assembly code. In turn, this makes it difficult to efficiently compile C code into DSP instructions.

Table 1. FIR assembly code for TI TMS320C25 DSP

address	object code	mnemonic	operand	comment
003f	3c8e	LT	* ,AR6	; load new sample
0040	389a	MPY	*-,AR2	; multiply coefficient
0041	2080	LAC	*	; load summation variable
0042	ce15	APAC		; add previous product
0043	60af	SACL	*+,AR7	; save summation variable
0044	209d	LAC	*-,AR5	; move sample in memory
0045	609a	SACL	*-,AR2	

As an example, Table 1 shows a possible implementation of a finite impulse response (FIR) filter on a Texas Instruments TMS320C25 DSP.[3] The 'C25 architecture shows several special-purpose data registers. In contrast to RISC architectures, these registers are implicitly addressed in the assembly instructions, e.g., the "LT" instruction loads the T register, while the "APAC" instruction adds the contents of the P (product) register into the accumulator register.

The operand fields in the example denote indirect memory access operations via *address registers* (ARs). In most DSPs, address arithmetic can be performed in parallel to the central CPU data path by means of a dedicated *address generation unit* that supports complex addressing modes. For instance, the operand field "*-,AR2" denotes (1) a memory access via the "current" AR, (2) an auto-decrement of the current AR after the memory access, and (3) a switch to AR number 2 as the pointer for the next memory access. In order to make the most efficient use of a DSP, the assembly programmer needs to be aware of the optimal use of all machine registers for the intended algorithm. Due to the highly irregular application-specific instruction set, DSP assembly programming is generally considered a very time-consuming and error-prone task. Software development at the assembly level requires careful debugging with graphical design environments such as those shown in Fig. 8.

4.2 C programming and compilation

A compiler translates high-level programming language code into machine-specific symbolic assembly code. Since the features of the C programming language still allow for a relatively low-level programming style that may guide a C compiler to make the right translation choices, C is the preferred programming language for DSPs. Moreover, C is a very widespread programming language, and a lot of legacy code written in C exists. C may also serve

[3]Note that this is a non-optimized implementation for illustration purposes. If programmed optimally, the DSP can perform FIR computation asymptotically within n cycles per sample, where n is the number of filter taps.

as a high-level interchange format that enables a path to software implementation from more abstract architecture-independent DSP design environments such as MATLAB (MathWorks) or SPW (CoWare).

It is useful for a DSP software developer to understand some foundations of compiler technology in order to achieve the best results. A C compiler performs translation by means of a sequence of different passes. The major steps in compiling C into assembly are as follows [7].[4]

Source code analysis: The source code is analyzed with respect to correct syntax and semantics according to the rules of the source language. In case of errors, corresponding messages are emitted. Otherwise, a *parse tree* is generated that serves as an *intermediate representation* (IR) of the input program. Depending on the compiler, the IR may also be represented in the form of *three address code* or *data flow graphs*.

Machine-independent code optimization: The IR is optimized by means of standard techniques such as *constant folding*, *constant propagation*, *common subexpression elimination*, *dead code elimination*, and others. These are mostly aimed at removing redundant computations to optimize code quality before assembly code generation in the backend.

Code selection: This pass translates the machine-independent IR to machine-specific assembly instructions. Since C code or IR code can be mapped to assembly in numerous different ways, the code selector has to make sure that an optimized mapping is found. Code selection is usually performed by *tree pattern matching* based algorithms.

Register allocation: The symbolic variables and temporary results of a program should be kept in the CPU registers during their lifetime, since register access is generally much faster than data memory access. It is the primary task of the register allocator to assign the (usually large number of) symbolic values to (a small number of) physical registers.

Instruction scheduling: There exist different types of dependencies between generated machine instructions. For instance, instruction A might compute a value that is used as an argument in instruction B, so that A must be executed before B. Further dependencies may result from the *resource occupation* of instructions. The use of specific CPU resources may result in mutual exclusion of instructions during certain cycles. The scheduler has to ensure that instructions are assigned to a minimum number of execution cycles while meeting all dependency and resource constraints.

While C compiler support initially has been quite poor, resulting in inefficient code [8], code optimization technology for DSPs has become more mature in the past decade. On one hand, DSP-specific code optimization techniques beyond classical compiler technology have been developed that contribute to higher code quality in terms of code size and performance. For

[4] The first two passes are generally denoted as the compiler *front end*, while the latter three form the *back end*.

instance, *address code optimization techniques* [9] ensure optimal utilization of DSP address generation units (see Section 4.1) for pointer arithmetic.

On the other hand, under the recognition that classical DSPs are difficult compiler targets, DSPs have become more "compiler-friendly," i.e., they allow the reuse of well-proven and effective code optimization technology. Recent *very long instruction word* (VLIW) DSPs, such as the Texas Instruments C6x series, are an example of this trend. VLIW architectures usually show a general-purpose register file that enables the use of graph-coloring based register allocation [7]. The compiler can exploit *instruction-level parallelism* by means of *list scheduling* for sequential code or more advanced instruction scheduling techniques like *software pipelining* for loops [10]. As a result, the quality of compiled code approaches that of hand-written assembly code.

References

1. J. L. Hennessy and D. A. Patterson, *Computer Architecture - A Quantitative Approach*, Morgan Kaufmann Publishers, San Francisco, CA, 2003.
2. http://www.bdti.com/pocket/pocket.htm
3. http://www.ti.com
4. http://www.analog.com/processors/index.html
5. http://www.agere.com/enterprise_metro_access/dsps.html
6. http://www.st.com/stonline/prodpres/dedicate/st100/document/document.htm
7. A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers - Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.
8. V. Zivojnovic, J. M. Velarde, C. Schläger, and H. Meyr, *DSPStone-A DSP-oriented Benchmarking Methodology*, International Conference on Signal Processing Applications and Technology (ICSPAT), 1994.
9. R. Leupers, *Code Optimization Techniques for Embedded Processors - Methods, Algorithms, and Tools*, Kluwer Academic Publishers, Dordrecht, ISBN 0-7923-7989-6, 2000.
10. S. S. Muchnik, *Advanced Compiler Design & Implementation*, Morgan Kaufmann Publishers, San Francisco, CA, 1997.

Microcontrollers

Steven F. Barrett[1] and Daniel J. Pack[2]

[1] Department of Electrical and Computer Engineering
College of Engineering
University of Wyoming
Department 3295, 1000 E. University Avenue
Laramie, WY 82071-3295, U.S.A. steveb@uwyo.edu
[2] Department of Electrical Engineering
United States Air Force Academy
6236 Fairchild Drive, Room 2F6
USAF Academy, CO 80840-6236, U.S.A. Daniel.Pack@usafa.af.mil

1 Introduction

In this chapter we introduce the reader to the fascinating world of microcontrollers. We assume that the reader has no background in this topic. We begin by describing what a microcontroller is. We then proceed to describe the unique niche microcontrollers occupy as compared to the personal computer (PC). We then describe the systems that are commonly available on a generic microcontroller. We specifically do not discuss a specific brand of microcontrollers but rather describe systems common to most microcontrollers. We then discuss advanced microcontroller features such as digital-to-analog converters (DACs), real-time clock systems, liquid crystal display (LCD) interfaces, and other advanced features. We then discuss how to choose a specific microcontroller for a given application and the steps involved in developing an application. We conclude this chapter with an example based on an automated home control system to unify the system chapter concepts. In the references section we provide supplemental reading material on specific processors [1–11].

2 What Is a Microcontroller?

Virtually all computers have the same component systems. They are equipped with a central processing unit or CPU, a memory system, an input/output system, a clock or timing system, and a bus system to interconnect constituent systems. The bus system consists of an address bus, a data bus, and a control bus.

You are probably very familiar with a personal computer or PC. For this type of computer, the systems are housed within the computer case or are

connected to the computer via connectors mounted on the case. Some of the other systems comprising the computer are hosted on the motherboard or main printed circuit board within the computer enclosure. A microprocessor is the integrated circuit or chip on the motherboard that contains the computer's CPU.

For many applications, a moderate amount of local computer processing power is required. For example, the gas pump at your local convenience store needs some local processing power to process keypad inputs from the user to select the proper grade of gasoline, activate the pumps, calculate total gasoline cost, etc. A PC would not be a good choice for this type of application. This is an ideal application for a microcontroller. A microcontroller is a self-contained computer in a chip. It contains all of the constituent systems previously described within the confines of a single chip.

3 Microcontrollers Versus a PC — Why not Use a PC?

Why would anyone want to use a microcontroller when there exists a powerful PC that can be programmed to perform almost any computer-related task? Two main reasons for selecting microcontrollers over PCs are the cost and the size.

Over the years, microprocessors in PCs have received the media's attention due to the rapid and amazing technological advancements of the past two decades. Behind the glamour of the microprocessors, microcontrollers have had their own equally amazing revolution. Today, microcontrollers are the best-selling type of processor. Unlike the microprocessors, microcontrollers are inexpensive (tens of dollars as compared to hundreds of dollars, and most are under ten dollars). In many applications, a powerful PC that can perform a variety of tasks is not necessary, but what is needed is a processor that can carry out one specific job. For those applications, cost effective microcontrollers are ideal. Microcontrollers are available in 4-bit, 8-bit, 16-bit, and 32-bit varieties. For each type, one can find a number of different clock speeds with a wide variety of memory configurations and onboard subsystems, which provide flexibility for design engineers to select the best microcontroller for a particular task.

The other main reason for selecting a microcontroller over a PC is the attractive compactness of microcontrollers: a computer on a single chip. All microcontrollers come with built-in on-chip memory and multiple input/output interface features. Many microcontrollers are equipped with analog-to-digital converters, pulse width modulated signal generators, a sophisticated interrupt system, multiple serial and parallel input/output ports, and a flexible timer system. These microcontrollers are placed in embedded systems to control a variety of functions. For example, in today's automobiles, one can find microcontrollers controlling an anti-lock break system, a fuel injection system, a cooling/heating system, a cruise control system, shock absorbers, front panel

displays, a media system, and a navigation system using a global positioning system (GPS). The single chip computer allows design engineers to incorporate a computer in any device that requires some computational faculties. These applications include home appliances, navigational systems, medical equipment, security systems, and robots.

4 The Generic Microcontroller

In this section we discuss the architecture of a generic microcontroller. The information we provide is independent of any specific manufacturer. We have done this on purpose. For manufacturer-specific details, we refer the interested reader to the textbook selection list provided at the end of the chapter. In Fig. 1 we have provided the block diagram of a generic microcontroller. We would like to emphasize that all systems shown in the diagram are contained within the confines of a single integrated circuit package.

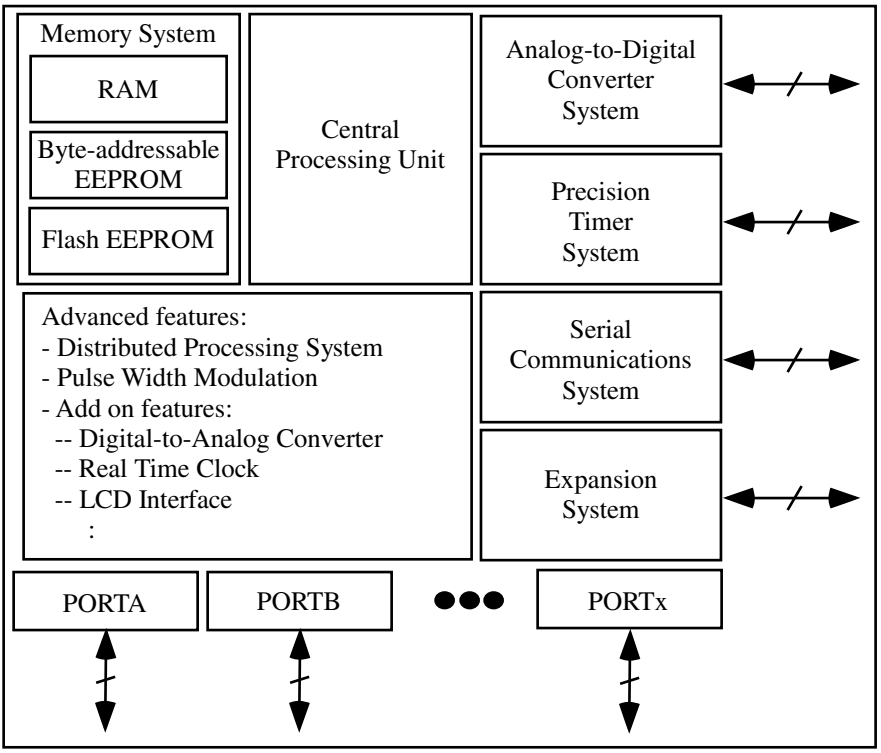


Fig. 1. Microcontroller block diagram

We discuss each system briefly in a clockwise fashion beginning with the memory system.

4.1 Memory system

As its name implies, the memory system contained within a microcontroller is used to remember the algorithm executed by the microcontroller, key program variables, and also system information.

A microcontroller's memory system is usually a conglomeration of different memory technologies. Most microcontrollers are equipped with a memory system containing both random access memory (RAM) and read-only memory (ROM) components. In an upcoming section we describe each type of memory and common applications for the memory type. However, we begin by discussing memory terminology.

Memory terminology

Memory capacity is usually expressed in terms of bits or bytes. A single memory bit has the capability of storing or remembering either a logic "1" or a logic "0" state. A byte is a collection of eight bits. It is a common method of expressing memory capacity. A memory that has a capacity of 1,024 bytes (referred to as a 1 kbyte) contains 8,192 bits of information.

Memory capacity does not give an indication of how the memory system is configured. For that, we must know the length and width of memory. Memory length indicates how many separate addressable locations are contained within the memory system. Memory width indicates the number of bits that may be stored in each memory location. For example, a 1,024 byte capacity memory could be arranged with a length of 1,024 locations with the capability to store a byte of information at each location. A 1,024 byte memory could also be configured to have a length of 512 locations with two bytes of storage capacity at each memory location.

The memory system is connected to the CPU by three different buses: the address bus, the data bus, and the control bus. A bus is a collection of conductors with a common function. The microcontroller's address bus provides a separate and distinct address for each memory location. Using binary address encoding, the address bus width (m) determines the number of distinct memory locations that can be accessed by the microcontroller. In general, 2^m specifies the number of separately addressable memory locations (M). The number of data lines (N) determines the memory width, e.g., one data line for each memory bit at each memory location.

A memory component is also equipped with various control lines. The read/write control line determines if the memory will perform a read or write operation. In the read operation the microcontroller extracts and uses the contents of a specified memory location. In the write operation, the microcontroller updates the contents of a specified memory location with new data.

The chip select line is used to select a specific memory component in a memory system containing more than one memory component. The output enable line controls when the memory component is allowed to place data on the data bus in a memory read operation.

A memory map is a tracking tool to describe the memory components connected to a microcontroller. As previously described, the number of address lines specifies the number of distinct, accessible memory locations. It does not mean that there are actual memory components present at each location. A memory map provides a visual display of the range of addresses accessible by the microcontroller and which specific addresses are populated with physical components. Memory addresses are usually specified in hexadecimal notation. A "$" sign preceding a number indicates a hexadecimal number. A sample memory map is provided in Fig. 2.

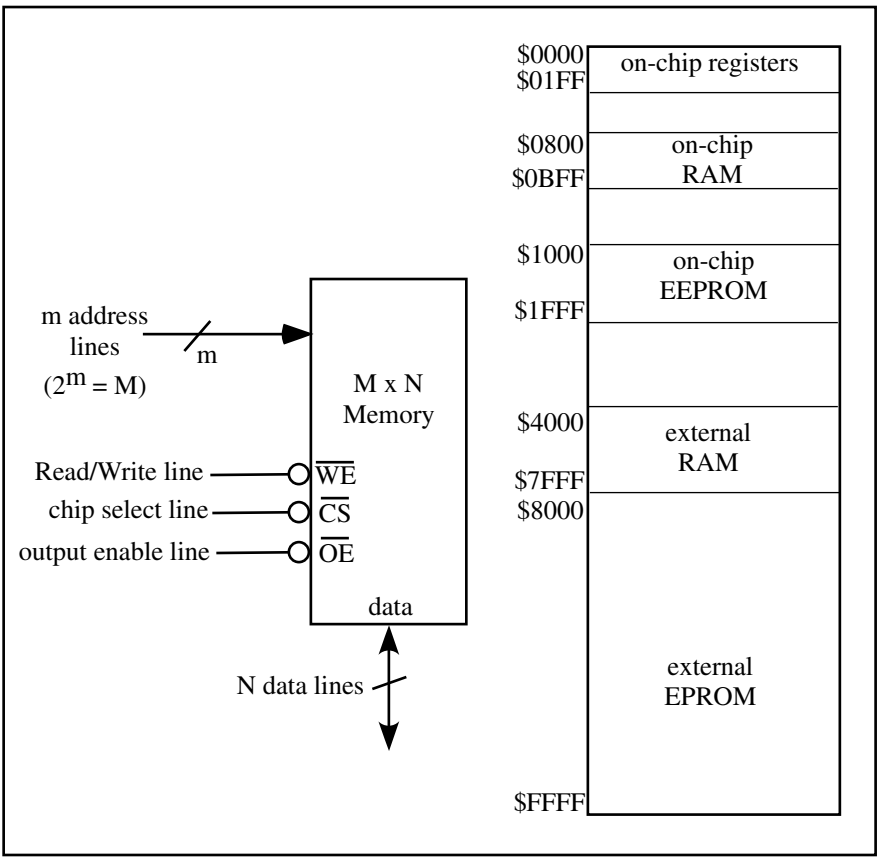


Fig. 2. Microcontroller memory map

Now that we have discussed basic memory terminology, let's take a closer look at the different types of memory components found in a microcontroller system.

"Flavors" of memory: RAM and ROM

As previously mentioned, most microcontrollers are equipped with a memory system containing both random access memory (RAM) and read-only memory (ROM) components. The RAM components are considered volatile. That is, when the memory component loses supply power, it loses its memory contents. On the other hand, ROM components are considered non-volatile. When power is lost, the ROM components retain their memory contents. Let's take a closer look at each type of memory component.

RAM

RAM configurations are used to hold program variables that might change during program execution. For example, in a higher order language such as C, global variables are stored in RAM. Therefore, RAM has the capability to be read from or written to. Usually RAM is used to store programs during algorithm development because they can be rewritten many times. This is an important capability since programs are changed considerably during development. Once an algorithm is finalized it is usually stored in some form of non-volatile ROM.

RAM is also used to host the processor's stack. The stack is a convenient storage location to temporarily store program variables during program execution. For example, when a function is called in a higher order language, any local variables within the function are declared on the stack. The return address to the main program is also stored here. Furthermore, during normal program execution, if a higher priority event occurs such as an interrupt, key processor register values are placed on the stack during interrupt execution and then recalled from the stack when the program returns to normal processing.

The RAM-configured memories typically have faster access times than ROM. That is, when the memory component is presented an address by the CPU, there is a shorter time delay (access time) to write to or read from the selected memory location.

ROM

ROM configurations are non-volatile, which makes them an ideal location to store a main program. That way should the microcontroller lose power, it will not lose its main program. There are many varieties of ROM. The different types are determined by how the contents of the ROM are originally placed in the memory. The types are ROM, PROM, EPROM, and EEPROM.

ROM is usually loaded with its contents at the factory. The programmable read-only memory (PROM) may have its contents written ("burned") into memory once. If the contents must be later updated, a new memory component must be used. An erasable programmable ROM (EPROM) may be programmed in the lab. If its contents need to be changed, the EPROM is placed in an ultraviolet light bath to zero out memory contents. The contents may then be rewritten.

The EEPROM, or electrically erasable programmable ROM, is available in two different varieties: byte-addressable EEPROM and flash EEPROM. Most microcontrollers are equipped with both types. Byte-addressable EEPROM, as its name implies, allows modification of single bytes of information during program execution. This type of memory is useful for storing program constants, security combinations, and fault status. Flash EEPROM may be rewritten in bulk. It does not allow for updating a single memory location. Flash EEPROM is used to store the microcontroller's algorithm.

4.2 Central processing unit

The heart of the microcontroller is the central processing unit or CPU. The CPU contains two main component parts: the arithmetic logic unit (ALU) and the control unit. The ALU performs the arithmetic operations (addition, subtraction, shift right, etc.) and logic operations (AND, OR, exclusive-OR, etc.) for the microcontroller.

The microcontroller is a synchronous state machine. It performs a sequence of operations, the fetch-decode-execute cycle of the CPU, in response to a system clock. To execute a program contained within its memory, it first fetches an instruction from memory. The instruction is then decoded to determine which operation is to be performed. The instruction is then executed. When the execution of the instruction is complete, the processor fetches the next instruction from memory and repeats the fetch-decode-execution sequence as illustrated in Fig. 3. This sequence of events continues until all instructions in the program have been executed.

An interrupt, as its name implies, is an interruption in normal program flow. An interrupt is usually associated with a higher priority event. In response to an interrupt, a processor temporarily suspends execution of the program, performs interrupt related instructions, and then returns to normal program execution.

4.3 Crystal time base

The time base for the processor is usually provided by a quartz crystal or a ceramic resonator. The quartz crystal provides a more accurate, stable time base. The typical maximum speed for a microcontroller is on the order of 1–10's of MHz. Most microcontrollers have the capability to vary system operational speed up to its maximum rated speed. The operational speed of the microcontroller is adjusted to the specific application.

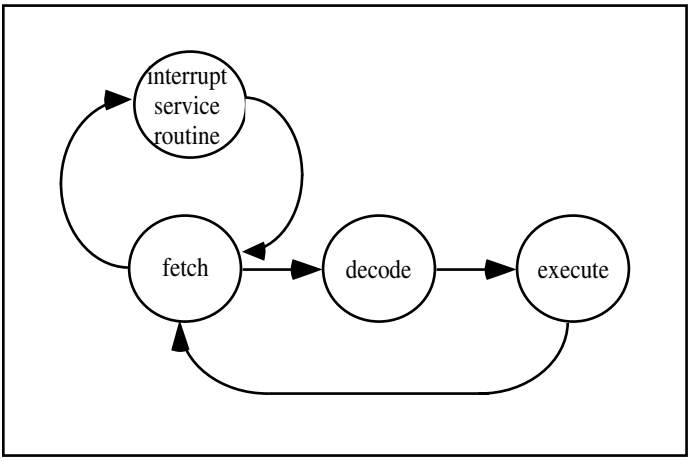


Fig. 3. Microcontroller instruction execution cycle

4.4 Analog-to-digital converters

A microcontroller is usually used in an application where it senses external physical parameters such as temperature, pressure, and light intensity. Based on the data it gathers it renders an appropriate control action. For example, a microcontroller could be used to control a fire sensing and alarm system. It would monitor ambient temperature and also the presence of smoke. Should a hazardous combination of temperature and smoke be detected, the microcontroller could activate alarms and sprinklers, and automatically dial the fire department.

These physical parameters of interest are analog in nature. The microcontroller is a digital device. Therefore, most microcontrollers are equipped with multi-channel analog-to-digital converters (ADCs). The analog input signals are converted to a weighted binary representation as shown in Fig. 4.

Background conversion theory

To convert an analog sample to a weighted binary value, three steps must be performed: determining the sample rate, determining the required resolution of the converter, and encoding the voltage sample into a weighted binary value.

Sample rate

Determining the sample rate for the converter means simply deciding how often the analog signal must be sampled to adequately represent the signal. You probably have an intuitive feel for this already. For example, if we were

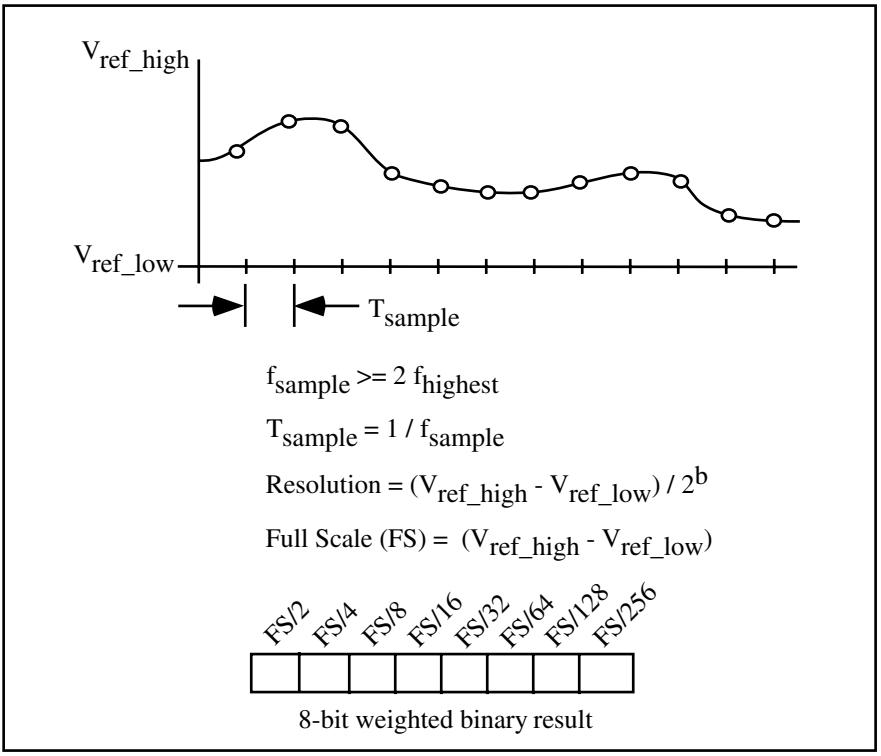


Fig. 4. Analog-to-digital conversion

sampling the outdoor temperature it would probably be adequate to sample it every 15 minutes. However, if you were sampling an audio signal in the range of 20–20,000 Hz, a much faster sampling rate would be required.

The Nyquist criterion establishes the connection between the sampling rate and the highest frequency content of a signal. The Nyquist criterion indicates that the analog sample must be sampled at a rate that is at least twice the highest frequency in the sampled signal. This can be expressed as

$$f_{sample} = 2\, f_{highest}.$$

The highest frequency content of a signal can be determined using various frequency analysis techniques. The interested reader is referred to the text by Bracewell listed in the references section at the end of the chapter [12]. Once the highest frequency of the analog signal is determined, a low pass filter (LPF) should be inserted before the analog input of the processor. The LPF, with a cutoff frequency equal to $f_{highest}$, prevents aliasing effects.

Basically, determining the sample rate means determining the time resolution of the sample. In a similar manner the voltage resolution of the sample must also be determined.

Resolution

The voltage resolution describes how finely an incremental change in the analog signal can be detected and recorded in the weighted binary representation of the sample. To provide a more refined weighted binary representation of the analog signal, additional binary bits are required. The equation which ties the different resolution factors together can be expressed as

$$resolution \ = \ (V_{ref\ high} \ - \ V_{ref\ low})/2^b.$$

In the equation $V_{ref\ high}$ and $V_{ref\ low}$ are the reference voltages provided to the ADC. The input analog signal must lie between these two reference values. External conditioning electronics may be required to ensure that this condition is met. The variable b is the number of bits of resolution provided by the ADC. Typical microcontrollers commonly are equipped with 8 or 10 bits of ADC resolution.

Encoding

Microcontrollers are equipped with various ADC technologies. We do not discuss the different varieties here. What they all have in common is that they convert a single analog sample into a weighted binary representation. The most significant bit of the result is one-half the full scale voltage, where full scale voltage is defined as:

$$full\ scale\ voltage \ = \ V_{ref\ high} \ - \ V_{ref\ low}.$$

The next bit represents one-fourth the full scale voltage and so on. The least significant bit represents the resolution of the converter. To convert the weighted binary value to a floating-point (real number) representation, the following conversion may be used

$$voltage \ = \ (weighted\ binary\ result/2^b)(full\ scale\ voltage).$$

Data rate

The concepts of sampling rate and voltage resolution may be tied together with data rate. Data rate indicates the amount of information that is generated during the analog-to-digital conversion sequence. It is expressed as

$$data\ rate \ = \ f_{sample}\ b.$$

Most microcontrollers are equipped with 8 to 16 channels of ADC conversion capability. In the next section we take a detailed look at the precision timing system.

4.5 Timing system

Most microcontrollers are equipped with a multi-channel precision timing system. The timing system has a variety of precision timer functions including measuring the parameters of an incoming digital signal, generating a precision output signal, or counting events.

Digital signal parameters

The common parameters of a digital signal are its period, frequency, and duty cycle. These parameters are illustrated in Fig. 5. The period of a repetitive (periodic) digital signal is determined by measuring the elapsed time between consecutive rising (or falling) signal edges. The frequency is not measured directly but may be calculated by taking the reciprocal of the period. The duty cycle is a measure of the percentage of the total period for which the signal is logic high. To measure duty cycle, the elapsed time between a rising edge and the falling edge must be measured as well as the period. The duty cycle expressed as a percentage may be calculated as

$$duty\ cycle\ =\ (on\ time/period)(100\%).$$

The free running counter

The main component of the timing system is a free running binary counter. The counter increments for each incoming timing pulse. The counter counts continuously from 0 to $2^b\ -\ 1$ where b is the number of bits in the counter. When the counter reaches its maximum count, it resets to "0" on the next incoming counter pulse. At the same time an overflow flag is set to indicate the counter has rolled over back to "0."

The free running counter usually has some method of resetting the counter to "0." However, this may not be a good idea if multiple timing channels are simultaneously using the counter. Since it is not advisable to reset the counter, elapsed time may be calculated using the following formula:

$$elapsed\ time\ =\ (stop\ time\ -\ start\ time) + (number\ of\ overflows)(2^b - 1).$$

The unit of elapsed time will be in "clock ticks." To convert to seconds the elapsed time in clock ticks must be multiplied by the period of the free running counter's time base.

Precision timer applications

As previously mentioned, a precision timer may be used to measure the parameters of an incoming signal or signals, generate a precision output signal or signals, count events, or perform some combination of these parameters. Each of these functions is briefly described.

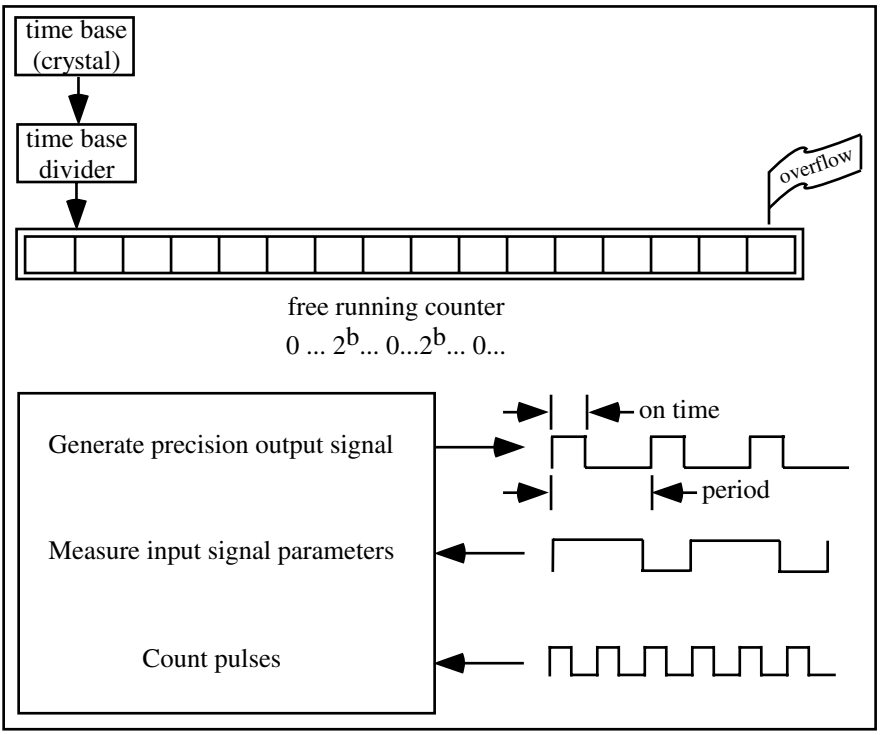


Fig. 5. Precision timer

Measuring parameters of input signals

The key parameters of an input signal that may be measured are period, frequency, and duty cycle. The parameters are measured by configuring a timer channel to log the count of the free running counter when certain signal parameters (rising and falling edges) occur.

Generating precision output signals

To generate an output signal, the desired signal parameters must be converted to clock ticks. For example, if a 1 kHz (period = 1 ms) digital signal with a 10% duty cycle is to be generated, the period and high time of the signal must be converted to clock ticks. If for example, the free running counter's clock source has a frequency of 100 kHz (period = 0.01 ms), the period of the 1 kHz signal to be generated will be 100 clock ticks and its high time will be 10 clock ticks. An algorithm may then be written to initially set a timer system output pin high, wait 10 clock ticks and then set it low, wait another 90 clock ticks and set it high, and then loop to continue generating the periodic signal.

Counting events

A precision timing system may also be used to count events. For example, if we equipped a rotating motor with an encoder that provided "n" number of pulses per motor revolution, we could develop an algorithm to determine motor speed in revolutions per minute (rpm).

4.6 Serial communications system

All microcontrollers are equipped with one or more serial ports to communicate with external devices. In fact, a microcontroller uses one of its built-in serial communication systems to allow a controller designer to communicate with the microcontroller using a PC. The communication involves downloading programs to be executed during system development, uploading data from a microcontroller to be viewed by a user, and interactively interfacing with the microcontroller using what are called monitor commands.

Such communication systems are necessary to provide microcontrollers the capability to transfer data to, monitor the status of, and control the states of external systems. Time-critical interfaces with external systems are usually accomplished through the microcontroller's parallel ports, which we discuss in an upcoming section. In this section, we confine our discussion to the serial interface. Typical microcontrollers have two different types of serial communication subsystems on board: one or more asynchronous communication systems and one or more synchronous communication systems. We briefly discuss these systems next.

Asynchronous communications

As the name indicates, an asynchronous communication system uses a stringent protocol to communicate with other serial communication systems. The most widely used asynchronous communication technique is the RS-232D (RS stands for Remote Standard) interface. We do not have space to fully describe the hardware requirements and software protocols here. The interested reader is referred to the textbook by Horowitz and Hill listed in the references section at the end of this chapter [13]. Data is transferred using the ASCII (American Standard Code for Information Interchange) standard or the newer international Unicode coding standard between two serial communication equipped systems.

The serial communication can be performed in the simplex mode, which allows one direction of communication at a time, or the duplex mode, which allows two-way communication simultaneously. Microcontrollers may use the duplex mode to interface with external devices.

To protect the integrity of the data transferred, the software protocol of the RS-232D method requires data to be transferred in a frame that contains data bits (8 or 9 bits), a start bit, and a stop bit. It also specifies the communication

rate, which is called the baud rate (bits per second), to ensure that the bit transmit rate matches with the bit receive rate. Compared to the synchronous communication interface, the asynchronous communication interface is slower, but it offers robust and cost (hardware) efficient communication when two systems are physically separated by less than 15 meters.

Synchronous communications

Unlike the asynchronous communication systems, a synchronous communication system requires that the communication parties use the same signal to synchronize participating parties. The synchronization can be obtained by issuing a synchronization pulse or, as is typically done, by using a common clock signal. The use of a common clock allows the data transfer rate to be significantly higher than the ones used in asynchronous communication systems. The synchronous communication system of a microcontroller should be used when the external device to monitor or control is in close proximity to the controller or when the demand for data throughput of serial communication is high. For example, in some applications, additional memory, an LCD display, or an extra port is needed. Although an asynchronous communication system can be used for such tasks, a synchronous communication system is more suitable for such applications due to its speed. Thus, using the synchronous communication system, one can program a microcontroller to access external memory, display status using an LCD or seven segment LEDs, and interface with external devices through additional ports with a relatively high speed.[3]

To use synchronous communication systems, a designer must specify which device is in control and which one is following the lead. The one in control is called the "master" and the one who follows the order is called the "slave." The master designated device is responsible for generating the synchronization signal or the common clock signal as well as the control signals to transfer data among the communication parties. Shift registers are used to send and receive data in synchronous communication systems.

4.7 Expansion system

As mentioned, the microcontroller is a self-contained system on a chip. However, should we need to add additional memory to the processor or additional systems not contained within the chip, expansion features are required. Most microcontrollers are equipped with an expansion port to provide external components access to the microcontroller's address, data, and control buses.

[3]If optimal speed is desired, one should use parallel ports with associated control signals. If serial communication methods must be used, one should choose the synchronous communication method if speed is the first priority.

4.8 Port systems

One of the main reasons for the success of microcontrollers in the processor market is the versatility of input and output interface options that microcontrollers offer. For example, the automobile industry pushed the application of microcontroller technology for systems employing multiple data collection sensors to make cars safer, fuel efficient, and robust. Another example can be seen in the robotics community. The availability of input/output ports is ideal to connect multiple sensors and actuators to control robot movements.

Typical microcontrollers have serial ports, for an asynchronous system and a synchronous communication system, and multiple parallel ports. Most of the ports are programmed to function for specific tasks such as analog-to-digital conversion or pulse width modulated signal generation. These same ports may be alternatively programmed to work as general-purpose digital input/output parallel ports. Fig. 6 shows a diagram of a typical microcontroller with its port specifications.

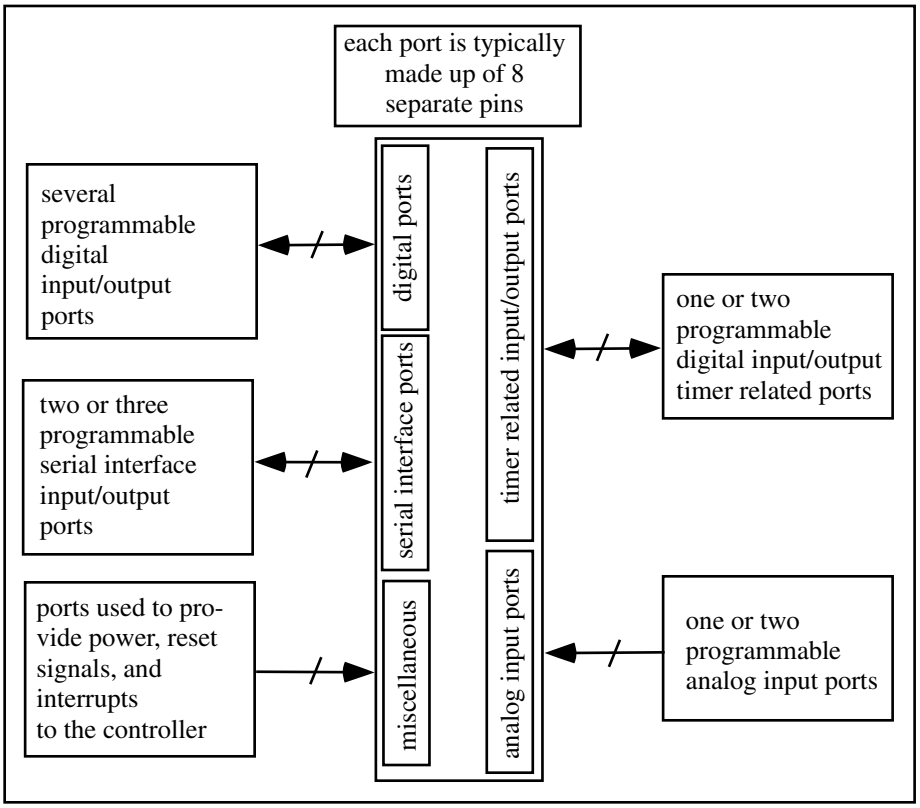


Fig. 6. Input/output ports for typical microcontrollers

4.9 Advanced features

In this section, we describe features that are designed for advanced applications. These features are not incorporated in basic, limited feature microcontrollers. However, due to the increasing number of applications benefiting from the advanced features, more and more microcontrollers are equipped with advanced features, some of which we discuss in this section.

Distributed processing

In this section, we present the capabilities of microcontrollers to connect to and communicate with multiple independent, distributed microcontrollers in a network. Such networks offer enormous advantages over an isolated system such as the capacity to share resources and data. In the microcontroller community, there are two different types of networks: Controller Area Network (CAN) [14] and a network based on the Byte Data Link Controller (BDLC) [15] that uses the Society of Automotive Engineers (SAE) J1850 protocol.

Both network architectures and protocols originated from the automotive industry during the mid-1980's when multiple microcontrollers were connected together to enhance automobile performance. Today, the two network architectures and protocols are found in audio systems, home theaters, communication systems, military systems, and some home appliances in addition to the car industry. We briefly point out the two methods and refer the interested reader to the reference section of this chapter.

The latest CAN protocol used in the CAN network has two different parts: part A and part B. Part A is made up of the Object Layer, Transfer Layer, and Physical Layer of Open Systems Interconnection (OSI) Reference Model. The CAN protocol part B consists of the Data Link Layer and the Physical Layer. The strength of the CAN protocol is the absence of originating or destination addresses for each message. Instead, an identifier is embedded in a message and each controller in the network is responsible for determining the receipt of messages by deciphering the headers of the messages. The advantages of such a scheme are (1) one can add a controller to a network without stopping the operation, and (2) the network allows multi-casting capabilities.

A BDLC-based network is useful for serial data communications at low speed, i.e., less than or equal to 125 kbps. It uses a variable pulse width bit format, noise filters, collision detection mechanisms, and cyclical redundancy checks to accurately transfer messages in a network. Over the years, both the SAE J1850 protocol and the CAN protocol have competed to dominate controller area networks. Currently, due to its speed and flexibility, the CAN protocol is gaining more popularity among industry users, poised to control the entire field of controller area network applications.

Pulse width modulation

Pulse width modulation is a method of precisely adjusting the effective voltage provided to an external component, as illustrated in Fig. 7. This is a common method of controlling the speed of a DC motor. For example, if we would like to control the speed of a motor with a required VDC supply voltage, we can send a signal of different duty cycles to the motor to precisely control its speed. The effective voltage delivered to the motor will be

$$V_{eff} = (VDC)(duty\ cycle).$$

Using this technique we can linearly adjust the speed of the motor from 0 to VDC. It should be noted that a microcontroller does not have sufficient voltage or current capability to drive a motor directly. An interface device must be used between the microcontroller and the motor. Reference Barrett and Pack [2] for a full treatment of this topic.

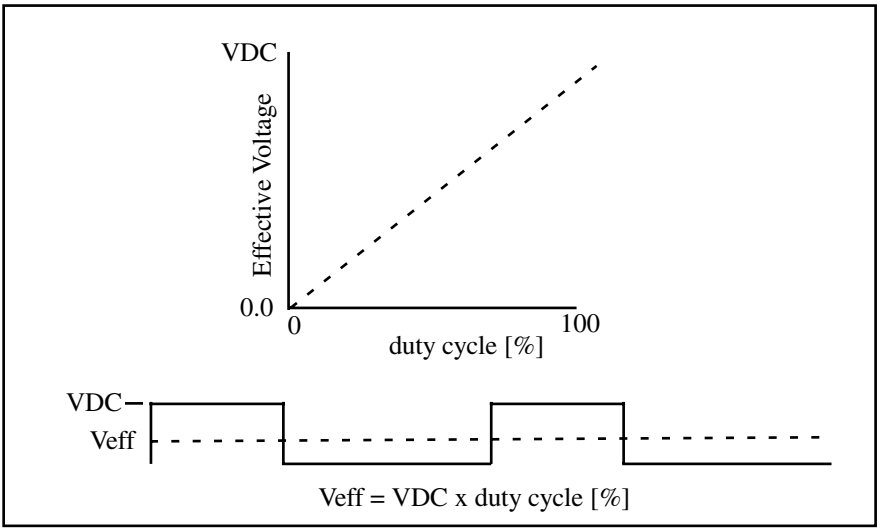


Fig. 7. Pulse width modulation

Add-on features

As mentioned earlier in this chapter, in many applications, the resources on a single chip microcontroller are not sufficient, requiring the use of external devices. We present a few such sample devices next.

Digital-to-analog converter

Many microcontrollers have embedded ADCs which are used to bring in analog signals for signal processing.[4] The digital-to-analog converters (DACs) become important once digital signals have been processed to generate analog signals for external devices. DACs are interfaced with microcontrollers in a number of different ways. The most prominent method used is through the serial port of a microcontroller. In particular, a microcontroller works as the master and, through a serial interface, controls and sends the digital signals to a DAC. Using this technique, multiple DACs may be interfaced to a microcontroller.

Real-time clock

In some microcontroller-based applications, external real-time clocks are connected to microcontrollers to govern the activities of the controllers. The real-time clock features also allow the microprocessor to easily keep track of events in standard clock time of hours, minutes, and seconds.

Liquid crystal display (LCD) and seven segment light emitting diode (LED) interfaces

To display the status of an internal microcontroller state, LCDs are commonly used. These LCDs contain their own resident microcontroller that controls the timing and display functions. Again, the system microcontrollers work in a master mode while the microcontrollers in LCDs run in a slave mode.

Another common add-on device for display purposes is the seven segment display unit. These units are connected to a parallel port of a microcontroller that sends explicit signals to turn on a set of LEDs through a set of buffers. Instead of burdening a microcontroller to find the set of LEDs that correspond to messages, a decoder chip can be used to turn on appropriate LEDs.

5 Microcontroller Selection

How do you select a specific microcontroller for an application? To get started you should develop an interface diagram which shows all input and output components that will be connected to the microcontroller for a specific application. The diagram should clearly account for all connections made to the microcontroller. As part of this exercise you should also decide which specific microcontroller systems are required by the specific application. This will allow you to choose the microcontroller with the features you need. We will provide an example of the interface diagram later in the chapter.

Here is a summary of questions you should answer to aid in selecting a specific microcontroller:

[4] Usually some sort of signal conditioning is done before the analog signal is sent to an ADC.

- Which specific microcontroller systems are required by the application?
- What is the time resolution required by the overall system? This will help determine an appropriate operating frequency for the processor.
- What size of numerical argument will be processed by the microcontroller? This will help you decide between a 4-bit, 8-bit, or 16-bit processor.
- How many digital input, digital output, analog input, and analog output channels are required by the specific application?

5.1 Availability — Who manufactures what?

Microcontrollers are produced by a wide variety of international manufacturers. Here is a partial list (in alphabetical order): Advanced Micro Devices, Atmel, Dallas Semiconductor, EM Microelectronic-Marin SA, Intel, Microchip, Motorola, National Semiconductor, Parallax, and Zilog. We will not attempt to describe the microcontroller product lines available from these manufacturers. For complete information on a specific product line we recommend that you visit the website for each manufacturer. You will find a plethora of up-to-date technical data, applications notes, and other helpful information. Here we describe how to match a specific processor to a specific application.

6 System Development

In this section, we describe some key issues associated with system design. The key for good system design is based on structured design methods. Simply put, structured design implies that the task of creating an embedded system is systemically divided into organized groups of small, minimal tasks. To aid such efforts, the design engineering community developed "divide-and-conquer" techniques such as structure charts, activity diagrams, and top-down design/bottom-up implementation methods.

These techniques are also referred to as functional decomposition, structured design, structured programming, and stepwise refinement. By breaking a big project into small pieces, these techniques provide designers with a methodical design approach to convert a number of system requirements into specific implementation plans to meet these requirements. By adhering to structured design techniques, one can increase the likelihood of creating projects that are reliable, flexible, and easy to maintain.

Similar to a flowchart, the structure chart is a graphical description of all subsystems using hierarchical modules and arrows that describe the relationships among the modules. The structure chart is used to illustrate the overall components of an embedded system. An activity diagram functions as a flowchart when we use the Unified Modeling Language (UML). The activity diagram provides an overall view of program flow. Fig. 8 shows an overall structured design process using an activity diagram.

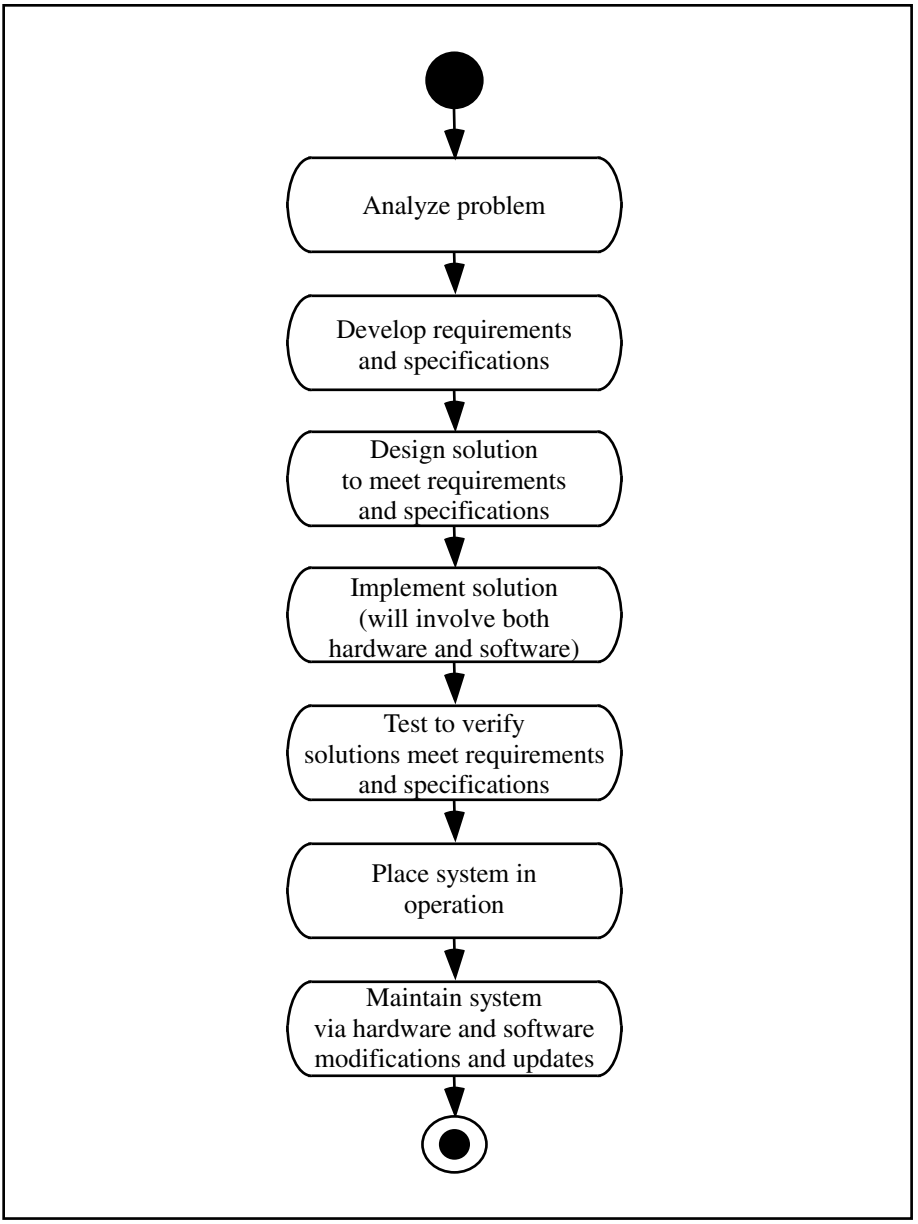


Fig. 8. The structured design process (adapted from the work of [2])

There are three different approaches to develop an embedded control system. The first approach is called the *top-down implementation* technique. To use the top-down implementation technique, the top module of a system is implemented first while functions of all the low level modules are simulated. Once the top module is implemented successfully, each module is implemented from the top to bottom until all modules are implemented. The second approach, called the *bottom-up implementation* technique, uses the opposite direction for system implementation. For the second approach, first all modules in the lowest level are implemented individually. Once successful, the modules on the second lowest level are implemented. The process continues until the top level module is implemented.

The final approach is a hybrid of the previous two. In this approach, modules on the top levels and the bottom levels are simultaneously developed and completed. The complete system implementation is done as modules from the top and the bottom meet in the middle of a structure chart.

As the overall system is implemented using one of the three approaches described above, one of the crucial aspects of structured system development is that each module must satisfy the module specifications. This portion of system development includes testing, debugging, and verification. In all cases, an integration and test plan should be made as a part of the overall project development plan.

In summary, the steps involved in system development are (1) conversion of requirements to system specifications, (2) partitioning of the entire system into hierarchical modules,[5] (3) display of the relationships among the modules using a structure chart and program flow using an activity diagram, (4) subsystem development, (5) subsystem testing, and (6) integration of subsystems and testing.

7 Example System: Automated Home Control System

In this section we will examine an automated home control system. The system will control the environmental, security, safety, and health aspects of a generic home. Our generic home is shown in Fig. 9. Our generic home has three levels. The first level contains the living room, kitchen, and dining room. The second level has four bedrooms. The last level is the finished basement area. It has a utility room equipped with a water heater and an environmental system to heat and cool the air within the entire home. The basement area also contains a finished recreation area for playing ping pong, billiards, or watching videos. The home also has a two-car garage.

The home is on a small parcel of land that is landscaped with shrubs and trees, as illustrated in Fig. 9. A privacy fence surrounds the backyard. The

[5]Each module represents a subtask of the system. The hierarchical display of modules illustrates the relationships between modules.

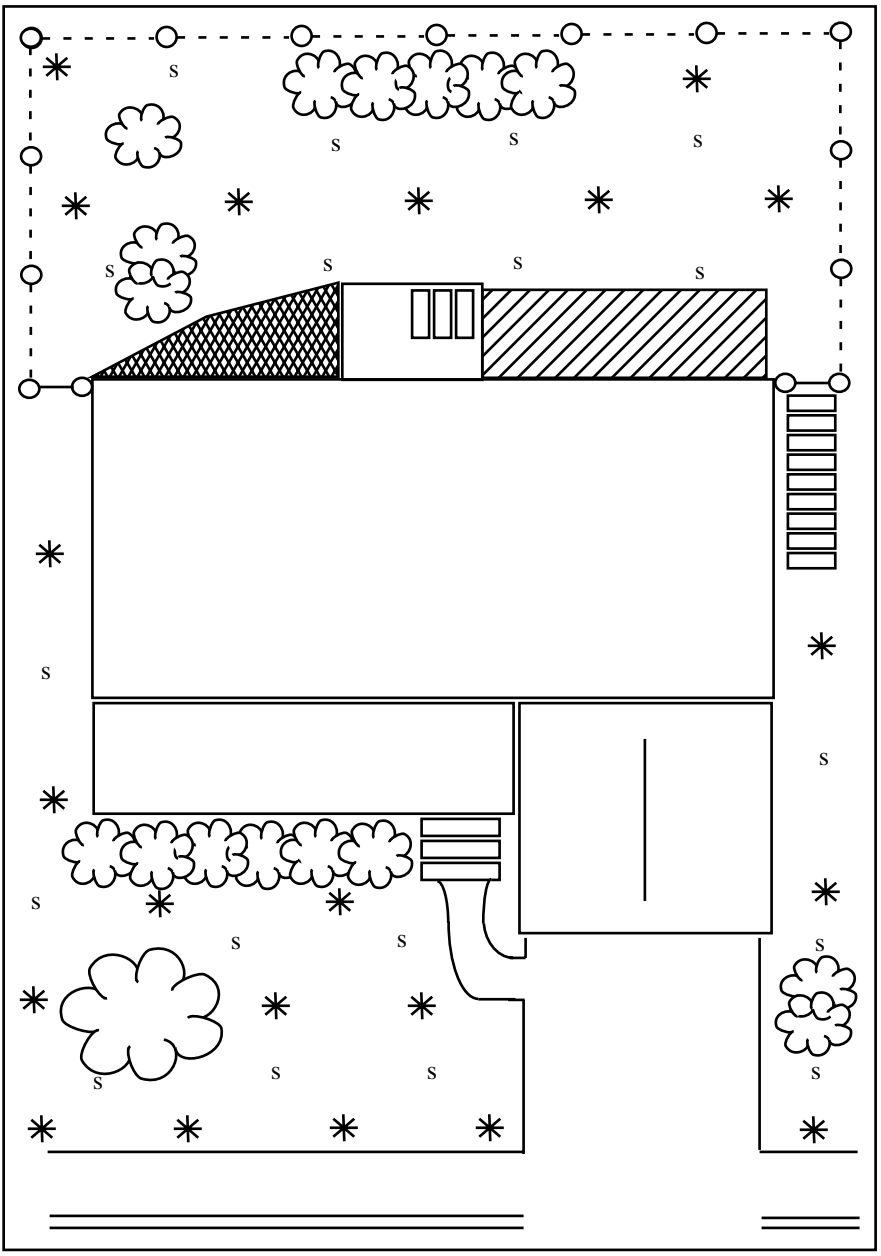


Fig. 9. Component layout of home control system

lawn is equipped with a water sprinkling system. A "*" indicates the location of a sprinkler head on the lawn. An "s" indicates the location of a moisture sensor on the lawn.

Overall, the microcontroller system will monitor multiple channels of data, assimilate the data via a user-developed control algorithm, and then assert different activities. Our control system could be rendered with a single processor or a distributed system of processors configured in a CAN. We will investigate a single processor solution.

Here are the different desired system features:

Maintenance. The control system will be equipped with sensors to measure the moisture content of the soil. When the soil sensors detect a dry lawn condition as specified by the homeowner, a multi-head sprinkler system will be activated to water the lawn, trees, and shrubs. Twenty one sprinkler heads (*) are required to adequately cover the lawn. The lawn is divided into three sprinkling zones. Each zone is equipped with six sprinkler heads. A separate 24 VAC control signal is required to activate the pump associated with a sprinkling zone. Twenty one sensors are required to monitor ground moisture. These sensors provide an analog output voltage signal from 0 to 5 VDC to indicate the moisture content of the ground from one extreme (arid: 0 V) to another (saturated: 5 V).

Environmental. The environmental system consists of a combination heating/cooling system to maintain different portions of the home at desired temperatures. Each of the three levels of the home contains four rooms. Each room contains a single sensor to monitor room temperature. The sensor provides an analog output voltage from 0 VDC (0 degree Fahrenheit) to 5 VDC (100 degrees Fahrenheit). The ducting system to deliver the conditioned air to each room is equipped with a damper control to adjust airflow to a specific room. Each of the twelve damper controls requires an analog signal from 0 VDC (damper closed) to 5 VDC (damper fully open).

Security. The home has a total of 24 doors and windows. Each of these is equipped with a simple magnetic reed switch to indicate if the item is open (logic "1") or closed (logic "0"). If the home is equipped with a single audible alarm, only a single digital output signal is required to activate a breach of security alarm for the entire home—monitor windows, outdoor zone security, motion sensors, video cameras, pet monitor.

Safety. The safety system consists of a series of smoke and carbon monoxide sensors placed throughout the home. Each level of the home is equipped with four fire alarms. These alarms provide a logic high output to indicate the presence of an alarm condition within a specific room. When an alarm condition is detected, a single home-wide fire alarm is activated. This home alarm requires a single logic high signal. Furthermore, each room is equipped with a water sprinkler system. The sprinklers are activated with a single logic high signal. The basement utility room is equipped

with a carbon monoxide (CO) detector. This detector is equipped with an analog sensor that provides a 0 VDC signal for no CO present to 5 VDC for harmful CO levels present. As the CO level approaches a harmful level a ventilation fan is activated in the utility room and a CO alarm is activated home-wide. Both require a single digital signal.

Communications. The communications interface provides a link between the microcontroller and outside agencies (police, fire department, etc.) via a telephone dial-up system. Telephone-compatible dual tone-multi-frequency (DTMF) tones are generated using a pulse width modulation system. The DTMF tones are the audible tones you hear when you depress telephone pushbuttons.

Central processing. The purpose of the CPU is to assimilate the collected data, make decisions via the algorithm coded by the user, and execute activities based on these decisions.

Interface. The homeowner will be able to interact with the control system via a 16-position keypad and an LCD. The keypad requires an eight-bit digital port interface while the LCD requires an eight-bit digital port and two additional digital data control lines. We assume that the user has options to activate different subsystems with the help of the keypad and the LCD. The interface allows the user to configure the home control system.

Backup power. Due to the critical nature of the control system, backup power will be provided by a standard off-the-shelf uninterruptible power supply (UPS).

7.1 Hardware interface diagram

The automated home control system may appear a bit overwhelming. We now provide some development tools to render a design description and list of requirements into a working prototype. As a start we have developed a hardware interface diagram, shown in Fig. 10. It illustrates each component of the home control system interface to the microcontroller. This is a good first step in choosing a specific microcontroller for the application. The hardware interface diagram clearly shows how many input and output connections are required for the specific application and whether they are analog or digital in nature. The diagram also illustrates any special microcontroller systems required by the application.

After constructing the diagram, you may not find a specific microcontroller to meet your needs. For example, the home control system requires digital inputs (44), digital outputs (28), analog inputs (31), and analog outputs (12). With this number of inputs and outputs, additional external circuitry will be required to multiplex the inputs into the limited number of pins on the microcontroller. Also, specially configured DACs that interface to the synchronous communication system may be required to provide multiple DAC channels.

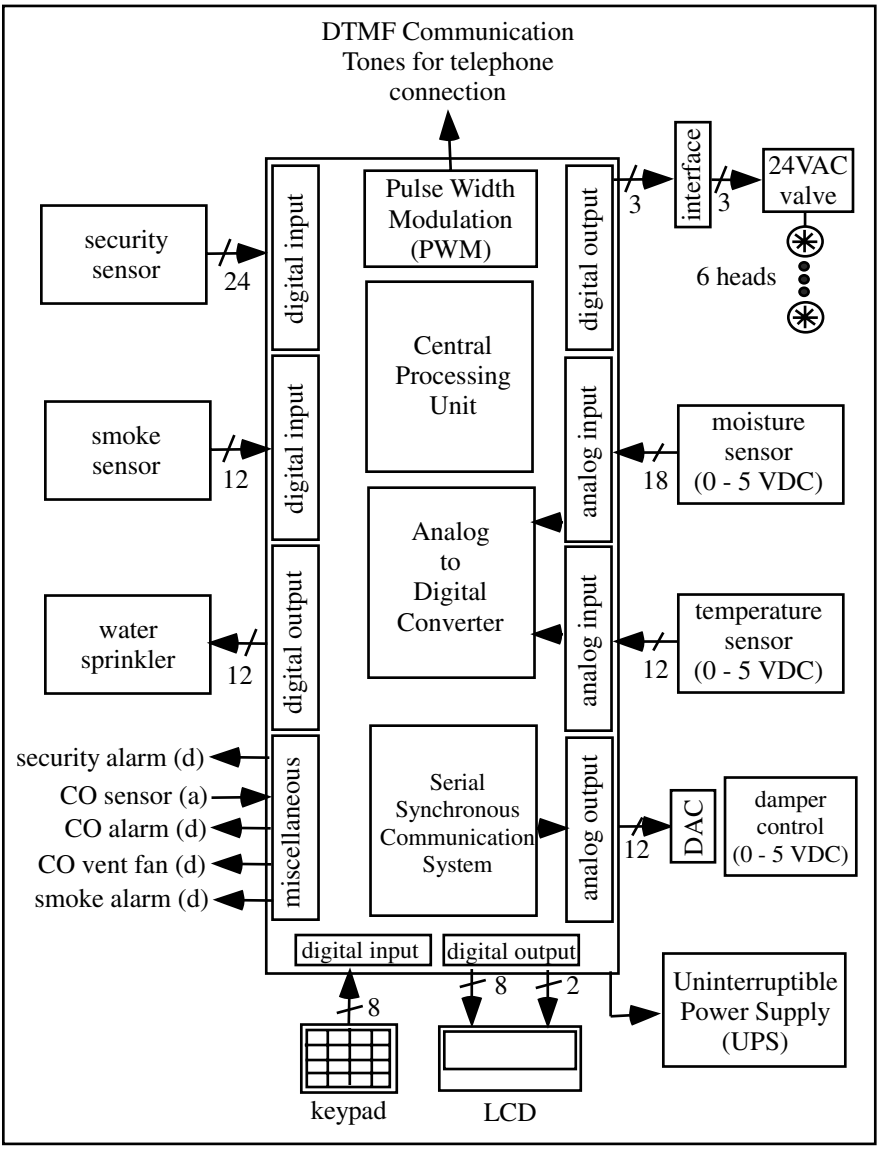


Fig. 10. Hardware interface diagram for the home control system.

7.2 System design

In this section, we design our home control system using activity diagrams. Fig. 11 and 12 show the corresponding diagrams for the main program and supporting interrupt service routines. We can implement the software based on

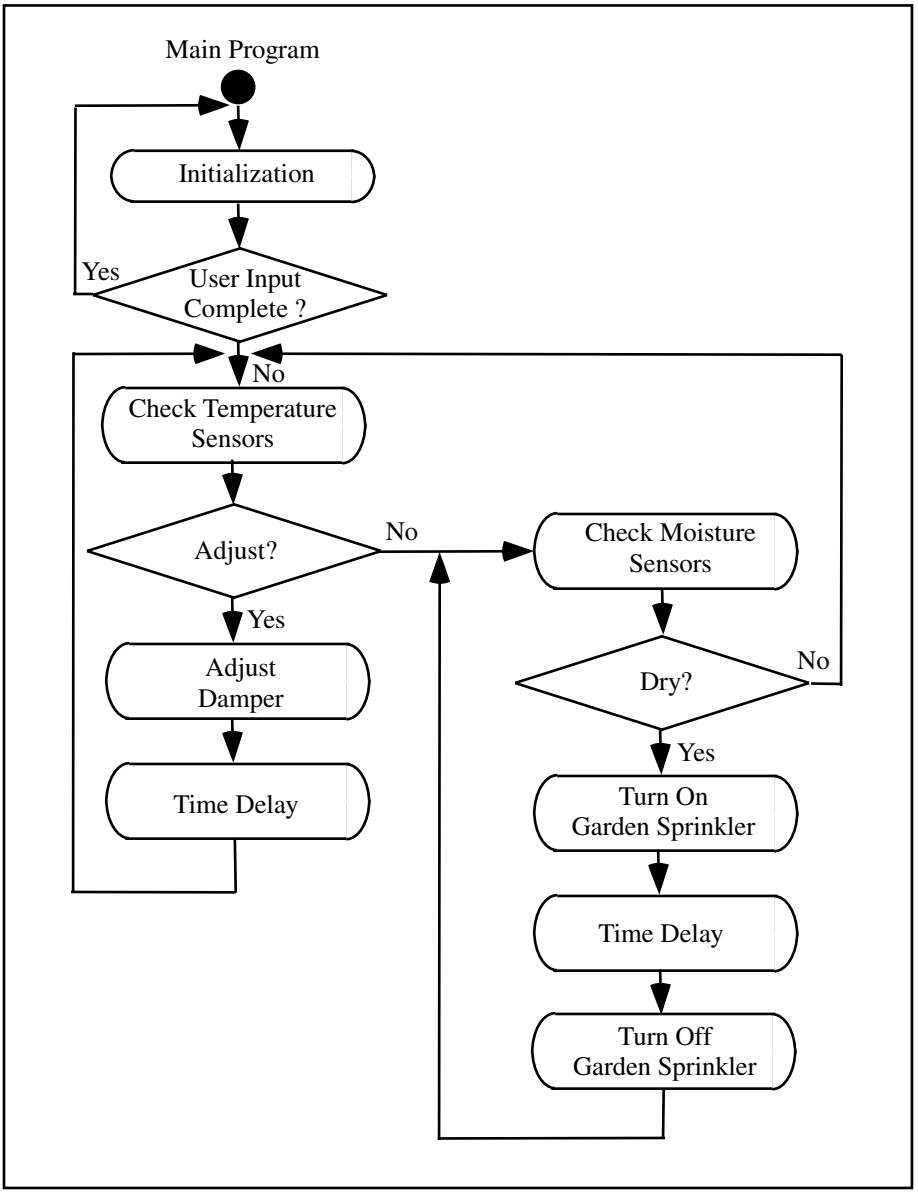


Fig. 11. Software design of the home control system main program

polling techniques or interrupt-based methods. We chose a hybrid approach where non-critical tasks (indoor temperature and watering the garden) are polled and the critical tasks (responding to high CO levels, a high smoke level, and a security breach) are driven by interrupts.

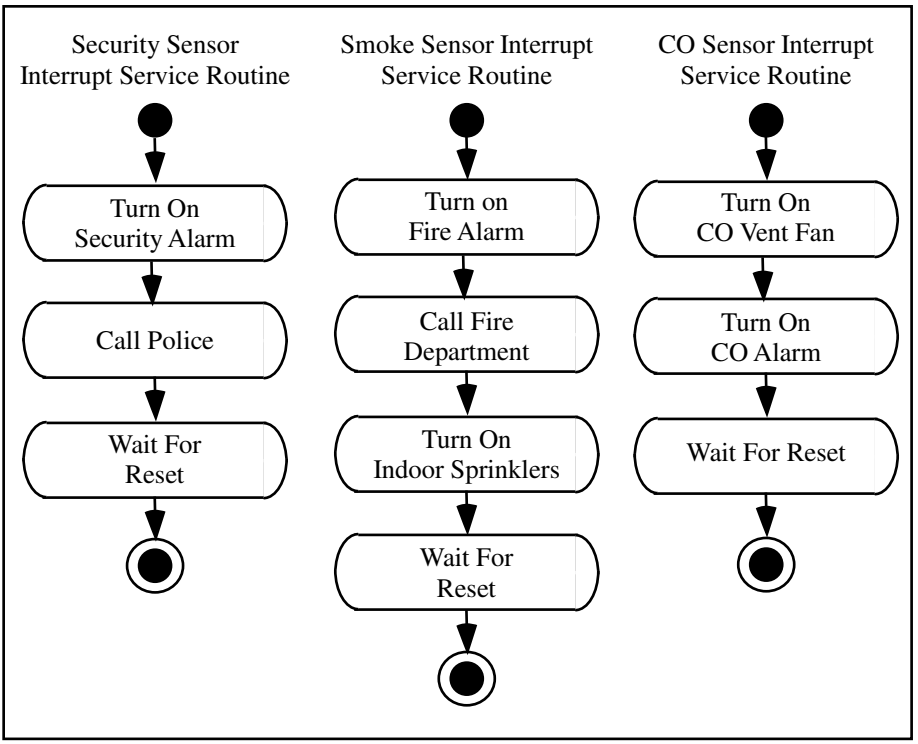


Fig. 12. Software design of the home control system supporting interrupt service routines

After initial configuration of the system, the polling portion of the software starts. We put the indoor temperature control at a higher priority than the watering of the garden. If an interrupt occurs, indicating that the system must service a task immediately, the system halts the current task and executes one of the three interrupt service routines selected. The time delay module shown in the polling portion of the software determines the time duration of dampers or garden sprinklers being turned on. The time duration is programmed during the initial configuration by a user.

To respond to a situation when more than one interrupt occurs, the interrupt priority should be programmed. Once an interrupt occurs based on sensor values, appropriate actions (turn on a fire alarm, a security alarm, a CO

alarm, etc.) are executed and the system waits for an operator to intervene. Once an interrupt occurs, the operator must restart the entire system.

Now that the activity diagrams and the hardware diagram are complete, designers can use the top-down or bottom-up approach to implement the system using a selected microcontroller. For implementation issues, we refer readers to the references section.

References

1. Pack, D. and Barrett, S. (2002), *68HC12 Microcontroller: Theory and Applications*, Prentice-Hall Incorporated, Upper Saddle River, NJ.
2. Barrett, S. and Pack, D. (2004), *Embedded Systems Design with the 68HC12 and HCS12*, Prentice-Hall Incorporated, Upper Saddle River, NJ.
3. Predko, M. (2000), *Programming and Customizing PICmicro Microcontrollers*, McGraw-Hill/TAB Electronics, New York.
4. Clark, D. (2003), *Programming and Customizing the OOPic Microcontroller: The Official OOPic Handbook*, McGraw-Hill, New York.
5. Predko, M. (1999), *Programming and Customizing the 8051 Microcontroller*, McGraw-Hill/TAB Electronics, New York.
6. Mackenzie, I. (1998), *The 8051 Microcontroller*, Prentice-Hall Incorporated, Upper Saddle River, NJ.
7. Gadre, D. (2000), *Programming and Customizing the AVR Microcontroller*, McGraw-Hill/TAB Electronics, New York.
8. Williams, A. (2002), *Microcontroller Projects Using the Basic Stamp*, CMP Books, Gilroy, CA.
9. Van Sickle, T. (2002), *Programming Microcontrollers in C*, CMP Books, Gilroy, CA.
10. Skroder, J. (1996), *Using the M68HC11 Microcontroller: A Guide to Interfacing and Programming*, Prentice-Hall Incorporated, Upper Saddle River, NJ.
11. Lipovski, G. (2000), *Embedded Microcontroller Interfacing for M-CORE Systems*, Academic Press, NY.
12. Bracewell, R. (2000), *The Fourier Transform and Its Applications*, Prentice-Hall Incorporated, Upper Saddle River, NJ.
13. Horowitz, P. and Hill, W. (1990), *The Art of Electronics*, Cambridge University Press, London.
14. CAN-Bosch Controller Area Network (CAN) Version 2.0, Protocol Standard, BCANPSV2.0/D, Rev. 3, Motorola.
15. BDLC Reference Manual-HC08 and HC12 MCUs, BDLCRM/D, Motorola.

SOPCs: Systems on Programmable Chips

William M. Hawkins

Department of Electrical and Computer Engineering, University of Maryland, College Park MD, 20742, U.S.A. bhawk@eng.umd.edu

1 Introduction

In this article we explore *systems on programmable chips* (SOPCs), that is, the concept of designing and implementing entire digital systems (processor, memory, and I/O, plus special functions and software) on the desktop, using inexpensive programmable chips and freely available tools. Before embarking on an SOPC test design, we first review the characteristics of the two main lines of programmable devices, *field programmable gate arrays* (FPGAs) and *complex programmable logic devices* (CPLDs), in the context of their relative advantages for different types of SOPC designs. We then give a brief overview of the primary enabling force for SOPC design, the *design framework*.

SOPC design encompasses the following elements:

- *Device*: usually classified under the major heading FPGA or CPLD, which in the pure forms can be thought of as arrays of logic elements with configurable interconnects, but which may also include devices with hardwired hardware functions such as *central processing units* (CPUs) or *random access memory* (RAM);
- *Tools*: an *integrated design environment* (IDE) that organizes the elements of a design; a set of point-tools that process different aspects of a design;
- *Intellectual property* (IP): component designs that will be interconnected to form the system design and may include such things as a CPU, numerical transform unit, memory, I/O devices, and also software such as operating systems and specialized algorithms. Such IP, whether open source or proprietary, is usually considered as black-box components to be integrated into the system design;
- *Software*: application programs running on the SOPC CPU;
- *Platform:* a circuit board, power supply, buffer chips, and whatever else is required to connect the SOPC to its environment.

2 Chips for SOPC Design

The selection of a programmable device for SOPC applications sometimes depends on external factors (e.g., tools, available IP, and platforms), but there are a number of inherent characteristics of the devices themselves which make them suitable or unsuitable for particular applications. Some of these characteristics flow from basic architectural features, while others reflect the intent of the device manufacturers to optimize their products for particular segments of the market.

The primary choice to be made in SOPC design is between FPGA and CPLD. Dividing the choice into just these two categories oversimplifies things a bit, as there are other devices such as antifuse FPGAs[1] which may be used for SOPC designs. Furthermore, a fair amount of architectural convergence has occurred between FPGA and CPLD. Thus, the clean division of FPGA and CPLD characteristics present in the early history of the device types has been replaced by mixtures of features in which an FPGA or CPLD flavor may dominate but a significant amount of silicon is used to add features of the competing type. Vendors of FPGA and CPLD devices suitable for SOPC design have narrowed down to the following short list: Actel Corp. [2], Altera Corp. [4], Lattice Semiconductor Corp. [5], QuickLogic Corp. [9], and Xilinx Inc. [18].

Bearing in mind the caveat on the ongoing convergence of FPGA and CPLD types, we will now describe the dimensions on which FPGAs and CPLDs are generally thought to differ.

2.1 FPGA

Fig. 1[2] shows a section of the block diagram for the Xilinx Inc. Spartan-IIE series of FPGAs [16]. Visible in the figure are:

- Input/output blocks (IOBs): latching, buffering, and level-translation for signals entering or leaving the chip;
- Configurable logic blocks (CLBs): fundamental building blocks for logic designs;
- Delay-locked loops (DLLs): clock scaling and phase control;
- Block RAMs: configurable (width and depth) dual port RAMs;
- Programmable routing: interconnect for all of the above.

[1] Antifuse or other *one-time programmable (OTP)* technologies are not considered here. OTP devices are available in high densities and speeds appropriate for SOPC, and may have distinct advantages in military and space applications requiring increased device hardening and reliability, but their lack of reprogrammability makes them an expensive choice for initial development of large scale SOPCs.

[2] Fig.1, Fig.2, and Fig.4 are based on or adapted from figures and text owned by Xilinx, Inc., courtesy of Xilinx, Inc.

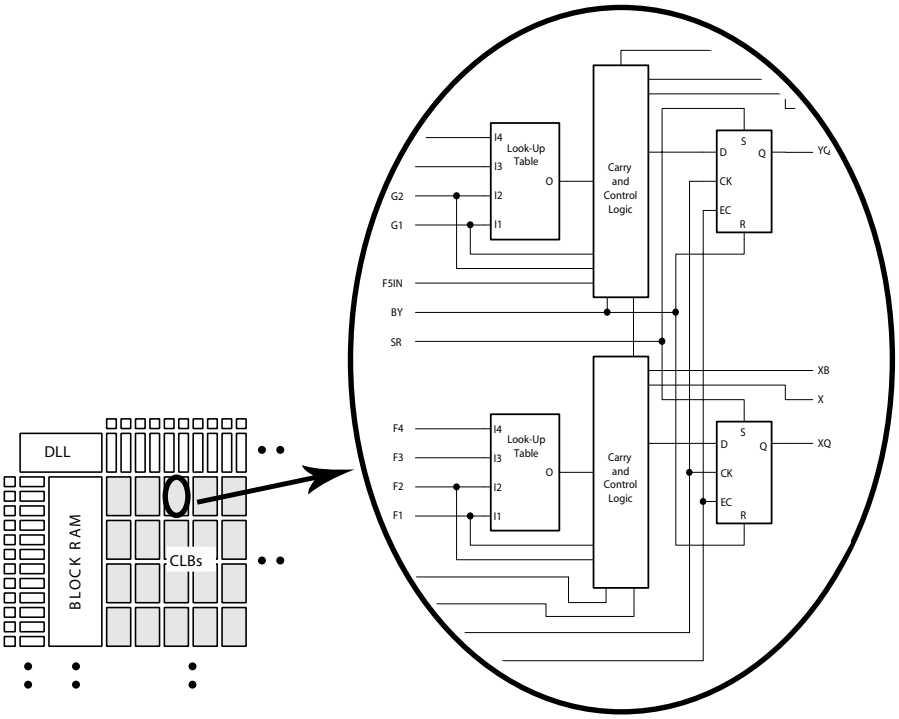


Fig. 1. Xilinx Spartan-IIE FPGA block diagram fragment with CLBs

The high-level view of an FPGA is that of an array of small logic blocks embedded in a grid of horizontal and vertical interconnection paths. Almost every hardware element in an FPGA has associated with it one or more bits of configuration RAM which is loaded when the chip is powered up or reset. Collectively, these bits store all aspects of the FPGA's operation; specifically, what logic function each element performs and how the elements are interconnected.

Individual logic blocks in the FPGA will contain several small (16 or so bits) RAM-based *lookup tables* (LUTs) which can provide any general logic function (4-in/1-out for a 16-bit LUT). The inputs of each LUT are connected through a RAM-configured connection array to local tap points on the interconnect grid. The LUT outputs are typically connected first through a RAM-configured arithmetic transform block which supports specialized functions such as "carry" for use in counters or adders, then through or around a RAM-configured flip-flop storage device which provides edge- or level-triggered storage of the LUT outputs or of the arithmetic transform, and finally, through a selection multiplexer to tap points on the interconnection grid.

The interconnection grid of the FPGA consists of a hierarchy of horizontal and vertical interconnects of different spans and connectivity. RAM-configured

connection arrays lying at the crosspoints of this grid provide the means for densely connecting signals to or from an associated local logic block and its immediately adjacent neighbors, using single-length connection paths. A second level of connection at the grid crosspoints provides for fanning signals out to a larger neighborhood, using connection paths originating at the connection array and continuing (with tap points) through several neighboring connection arrays to the wider neighborhood. The ultimate level of connection at the grid crosspoints provides the capability of driving or receiving signals from global lines which span the entire width and height of the chip.

Clock signals in FPGAs follow special paths paralleling the interconnection grid. They originate at a small number of specialized clock-input pins of the device, are extensively buffered for driving large fan-outs with minimum propagation delays, and have configurable connections to the storage device clocks of every logic cell.

The external interfaces of the FPGA usually consist of a ring of I/O cells at the periphery of the interconnection grid. I/O cells resemble the FPGA's logic cells but without the LUT logic. In addition to the I/O cell's connections to the FPGA's interconnection grid, there will be a collection of buffers with programmable pullups, pulldowns, keepers, level shifters, delays, and protection devices, etc., for making the I/O characteristics of the external pins compatible with a wide variety of different signaling standards and application requirements.

2.2 CPLD

Fig. 2 shows the component hierarchy of Lattice Semiconductor Corporation's 5000MX series of CPLDs [6]. Visible in the figure are:

- Chip-level structure
  - Global routing pool (GRP): global signal connection matrix;
  - Output sharing array (OSA): augments signal connection for very wide-input logic functions;
  - Multifunction block (MFB): multipurpose logic/RAM array;
  - System input/output blocks (sysIO): latching, buffering, and level-translation for signals entering or leaving the chip;
  - Phase-locked loop blocks (sysClock): clock scaling and phase control;
- Multifunction block (MFB) level—SuperWIDE Logic Mode (shown)
  - AND-gate array: 164, 68-input AND-gates;
  - OR-gate array: 64, 5-input (expandable) OR-gates at the AND-gate array outputs;
  - Macrocell array: 32 flip-flops and signal steering and conditioning logic at the OR-gate array output;
  - Alternative function modes (not shown): Single or dual-port RAM, first-in first-out (FIFO), content addressable memory (CAM).

- Macrocell slice—horizontal section through MFB in SuperWIDE Logic Mode
  - AND-gates: 5, 68-input AND-gates;
  - OR-gates: 2, 5-input OR-gates with expansion and carry inputs and outputs;
  - Flip-flop: D-flip-flop with preset, reset, clock enable, and input and output steering logic.

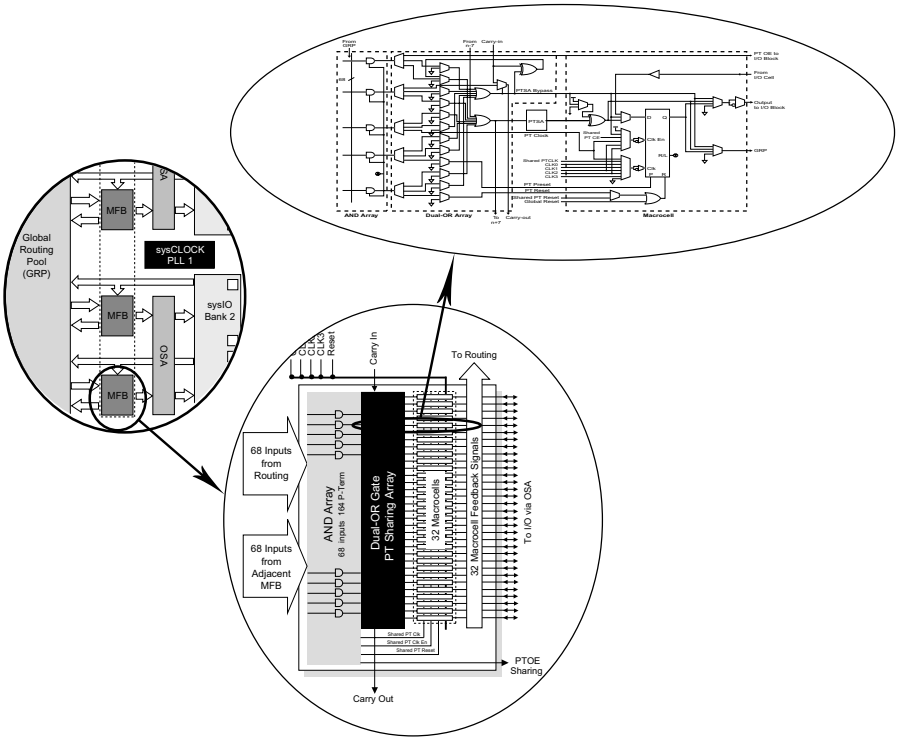


Fig. 2. Lattice 5000MX CPLD component hierarchy

The high-level view of a CPLD is that of a very large crossbar switch array (the GRP) with logic cells around the perimeter. The connection paths of the crossbar and the configuration of each logic cell are stored in *electrically erasable programmable read-only memory* (EEPROM) or flash memory cells when the device is programmed and are immediately available at power-on.

Individual logic cells of the CPLD typically contain a relatively small number of very large sum-of-products arrays which can receive their inputs from most or all of the crossbar outputs, that is, from most of the rest of the chip. At the outputs of the logic cell's product array are several stages of arithmetic

or logic transform and storage which resemble those following the LUT in an FPGA cell. The cell's outputs are steered through its output stages in a fashion similar to that of the FPGA cell, and the signal which finally emerges is sent back into the crossbar array and optionally to the device outputs.

CPLDs follow the same policies for clocks and I/Os that FPGAs do, that is, their clocks are treated as special signals and are carried on dedicated and highly buffered paths to the individual logic cells. CPLD I/O cells have specialized configurable pin drivers and receivers for interfacing to a variety of external signaling standards. One noticeable difference between CPLDs and FPGAs is that CPLDs usually employ an additional routing array between their internal logic cells and the output cells. This may arise from the CPLD's application advantage in the area of wide logic functions and reflect the need for an enhanced ability to collect the signals from a broader set of pins, or it may just be that the CPLD manufacturers are already very good at implementing crossbar switches.

2.3 Choosing a device type for SOPC design

The underlying architectural differences between FPGAs and CPLDs and our own priorities or those of the application may guide our choice of the most appropriate device. Some factors we should consider when comparing devices from different vendors are:

- Fine-grained versus wide fan-in;
- Onboard RAM;
- Special functions;
- Overall implementation cost;
- Startup latency;
- Power consumption;
- Design security;
- Dynamic reconfigurability;
- Architectural enhancements.

Fine-grained versus wide fan-in

Because FPGAs have a relatively large number of low fan-in[3] combinatorial logic and storage components distributed over a two-dimensional field which is rich in local interconnects, fine-grained design elements such as register arrays and counters are easily and efficiently implemented in FPGAs. However, whenever high fan-in logic such as an address decoder is to be implemented in an FPGA, it must be constructed from a "tree" of smaller structures. This both consumes additional resources to connect the tree structure and adds layers of propagation delay to the logic function. Conversely, the much higher

[3]The number of inputs to a logic device.

fan-in capability for the CPLD cell will usually eliminate the need to use more than one logic cell or require more than one layer of propagation delay in a decoder design, but the CPLD's relatively low number of storage elements will be quickly exhausted by a register-intensive design.

The makers of FPGAs and CPLDs attempt to reduce the penalties inherent in particular features of their respective architectures in their newer products by adding convergence features, such as the following:

- FPGA: richer sets of global and intermediate distance signal paths; specialized helper functions (carry, etc.) which better combine logic cell outputs in common high fan-in situations; specialized routing arrays around the routing array perimeter, for more flexible I/O pin placement.
- CPLD: increasing the ratio of logic and storage cells to crossbar switch area by partitioning the device into crossbar subarrays; connecting crossbar subarrays with higher-level crossbar arrays; adding specialized helper functions and routing arrays for connecting I/O pins.

Overall implementation cost

For reasons of history and technology, FPGAs generally, but not always,[4] require an external configuration subsystem which provides the configuration bits to be loaded into the FPGA's RAM configuration bits on power-up, while CPLDs typically hold their configurations in on-chip EEPROM cells which are immediately active when the chip powers up. The cost of the SOPC must include the cost of configuration. In the case of the CPLD this cost is zero, but in the case of a modern multimillion gate-equivalent FPGA a large and fairly expensive configuration ROM will be required. However, because SOPCs are rarely standalone despite their name, opportunities may exist for lower-cost configuration methods for the FPGA, such as loading configuration data over a network or from disk.

Onboard RAM

FPGA device technology traditionally confers a built-in advantage over CPLDs in implementing medium-size blocks of onboard RAM. This is a distinct benefit for most SOPC designs in terms of necessity (for local storage), speed (of onboard RAM versus external RAM), and cost (for lowered parts counts).

Special functions

Most vendors of devices suitable for SOPC designs have added special-function components to their high-end devices. In some cases the added components

[4] Actel Corporation's ProASIC devices are exceptions here, but Actel's architecture, although termed FPGA, is also quite different from the classical CPLD or FPGA architectures.

have wide applicability, for example, CPUs and arithmetic functions. In other cases the vendor is pursuing a specific market such as telecom with extremely specialized devices (e.g. ATM switches) which are not likely to be useful in unrelated design areas.

Startup latency

The time required by the FPGA for loading its configuration may be a substantial fraction of a second, while a CPLD typically becomes operational at power-on. This is another disadvantage of the FPGA's volatility. This distinction may be unimportant, as in the case where the SOPC is starting up in parallel with other elements in a larger system and is not the slowest starter in the group. In certain cases, however, such as when the SOPC provides the startup conditions for the larger system and is thus on the startup critical path, startup latency may become an issue. In some cases such as ultra low-power systems, where chip power is switched to extend battery life, startup latency may rule out the use of an FPGA entirely.

Power consumption

All other things being equal, the power consumption of the configuration bit (an EEPROM cell) in a CPLD will be lower than the power consumption of the configuration bit (a static RAM cell) of an FPGA. However, since there is almost nothing else equal, either in terms of the device architectures or the SOPC design's implementation in them, the power consumption of a particular chip can at best only be roughly approximated from the basic information on the chip datasheet. The most accurate result will be obtainable by fully implementing the design on both chip types and measuring power consumption on the bench. A less expensive approach will be to simulate the designs as implemented on either chip type to obtain power consumption estimates from the design tools.

Design security

Security issues encompass the prevention of design replication by simple copying, the protection of trade secrets and proprietary hardware designs in the chip from reverse engineering, and the protection of data traversing the chip when employed to implement encryption algorithms and the like.

The fact that the FPGA's configuration is loaded from an external device means that the device configuration is exposed to anyone with simple tools such as logic analyzers, and that the design may be copied simply by writing the captured bit stream to another device. A CPLD's configuration may also be read; however, this feature can be turned off when the CPLD is programmed and cannot be reversed unless the entire device is erased, thus protecting the CPLD configuration's security.

FPGA manufacturers address the security issue in several ways. First, they provide security against reverse engineering through obscurity—the location, arrangement, and functions of configuration bits in FPGA devices are usually proprietary and very difficult to deduce by casual data inspection and exploration. Second, FPGAs may include integral decryption engines which allow the configuration bit stream to be stored in the external device in encrypted form.

Dynamic reconfigurability

The fact that a CPLD's configuration is stored in EEPROM cells means that erasing and reprogramming its configuration bits will typically take a relatively long time (milliseconds), whereas the RAM configuration bits in an FPGA can be rewritten in nanoseconds. Furthermore, the CPLD's erasure is typically a bulk process determined by device economics and occurs over the entire chip at once. In contrast, the FPGA's design is not greatly changed if a means is provided to allow individual elements in the chip to be reprogrammed while preserving the configuration of the remainder of the chip. Device manufacturers have added reconfigurability to their chips in several ways. One method is to associate several stored bits with each configuration point, and select which is to control the configuration by an external signal to select different sets of configuration bits. This method is applicable to both CPLDs and FPGAs and may be attractive for certain specialized applications, such as those which switch between several hardware algorithms nearly instantaneously, but has the defect of increasing the device cost almost linearly with the degree of reconfigurability. A different approach, which favors FPGAs over CPLDs, is to allow reprogramming of subareas of the device, and to map SOPC subsystem elements which must be changed entirely within such subareas. Reconfiguration can then be relatively fast due to the smaller portion of the design which must be reloaded, and the mechanism for loading the new configuration is the FPGA's existing configuration support, which has already been paid for.

An application for the dynamic reconfigurability of SOPCs which does not involve the SOPC's functionality but is a useful aid during debug is the ability to insert (virtual) probes into the SOPC hardware. Traditional debugging of board-level digital systems involves observing intermodule signals with logic analyzers and oscilloscopes. SOPC designers may attempt to substitute simulation for much of this, but often the device does not do what the simulation says it should, and it becomes necessary to investigate intermodule signals on-chip. This can be done quickly and flexibly in the design environment by dedicating a few external pins as probe pins and connecting them to conventional instruments, and then connecting various on-chip signals to the probe pins by reconfiguring the device.

Architectural enhancements

Several trends dominate in new generations of products from traditional manufacturers of FPGAs and CPLDs. The first is the convergence of the logic architectures of the two main device types. As mentioned earlier, FPGAs are adding special functions and connections to address their deficiencies for fan-in intensive problems, while CPLDs are becoming more fine-grained to address their deficiencies for register-intensive problems.

3 Tools for SOPC Design

Since it encompasses every aspect—from concept to implementation—of a system's design, SOPC design requires a large variety of disparate tools. Mastering the tool chain for SOPC design is often the most difficult aspect of SOPC development.

Some problems of SOPC design tools are listed as follows.

- Much of the information about the essential component of SOPC design, the CPLD or FPGA, is secret and proprietary to the device's manufacturer.
- Many of the algorithms behind a tool's operation are in a state of theoretical development and academic research.
- Many of the best tools for particular aspects of SOPC design belong to third parties, were developed for other tasks, and have idiosyncratic user interfaces and non-standard or proprietary data interfaces.

The recognition of the problems inherent in mastering this tool chain has led to the development of *design frameworks* for SOPC design. The primary functions of a typical design framework are design input, organization, simulation, analysis, translation (to configuration files), and programming of the FPGA or CPLD.

3.1 Design framework approaches

Two major approaches arose in the early days of presenting a design in a design framework: 1) the design-structure presentation and 2) the design-process presentation. The first approach solved the problem that a design had a large number of components which were intricately interrelated in a hierarchy, by presenting the design as a tree structure that mimicked the component hierarchy. The second approach solved the problem that a design had to go through a complex series of steps between concept and device programming, by presenting the design by its location in the process flowchart.

3.2 Chip-vendor supplied design frameworks

The current state of design frameworks is exemplified by Xilinx Inc.'s Integrated Synthesis Environment (ISE) [17], which currently has the largest following in the electronic design automation (EDA) market for FPGAs [24]. Xilinx's ISE provides a typical contemporary FPGA design framework which recognizes that neither the structure nor the process view alone is adequate and therefore presents both. A version of the ISE provided by Xilinx in their free "Webpack" design environment will be used in the SOPC design example that follows.

Other FPGA and CPLD vendors have their own proprietary approaches to design frameworks—Altera Corp. has their Quartus II [3]; Lattice Semiconductor Corp. has ispLEVER [7]. Each recognizes in different ways that complex designs must be viewed simultaneously from a number of different perspectives. Since each realizes that requirement in a distinct way, the issue of skills and design portability is introduced—a vendor's design environment is tightly bound to the vendor's chips. It is impossible to change one without changing the other.

3.3 Third-party design frameworks

Another approach, which may alleviate the portability problem, and which is the only approach available for vendors not having proprietary design frameworks, is to use a general-purpose third-party design framework such as those available from Cadence Design Systems [27], Mentor Graphics Corp. [8], or Synplicity Inc. [13]. A possible added benefit to this approach is that these vendors' tools often can provide a path to the development of application-specific ICs (ASICs); hence, designs may be migrated from FPGA to ASIC as considerations of volume, speed, and cost change.

4 Design Framework Example: Xilinx's Integrated Synthesis Environment (ISE)

Fig. 3 shows a screen shot (annotations added) of the SOPC example to be developed subsequently, taken from the Project Navigator (PN) interface of Xilinx's Webpack 6.2i [17]. PN is the main interface into all of the functions of the ISE. The interface is divided into four "views" of a project: "Sources" (upper left), "Processes" (middle left), "Console (dialog)" (bottom), and "Reports/Edit" (upper right). These different aspects of the project presentation in the PN are described in the following paragraphs.

4.1 Sources window

Every element which is an explicit part of the design project will be linked to it in the Sources window. This includes not only the components of logic de-

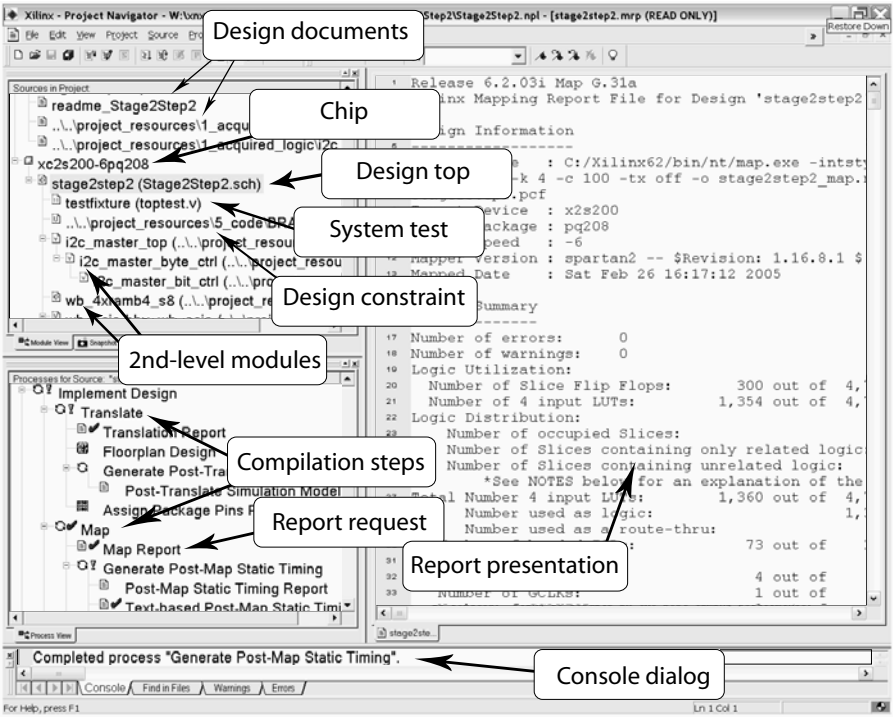


Fig. 3. Xilinx ISE Project Navigator

sign (marked as "Design top", "2^{nd}-level modules" in the screen shot) but also the identification of the target chip ("Chip"), testing procedures for the design ("System test"), constraints ("Design constraints"), and any supporting documents associated with the design ("Design documents"). Tiny icons next to each of the files in the Sources window indicate the source type (hardware description language, schematic, user constraint, etc.)

The design hierarchy presented in the Sources window resembles the folder view in most graphical file-system browsers. The top-level folder is the "Chip" folder, under which all design, constraint, and test elements are attached, and the "Documents" folder for items such as readme's which are not directly incorporated into the design.

The Chip folder's tree-structured organization is provided automatically by the ISE, which "understands" the relationships between everything stored there and depicts those relationships in the subfolder tree. The primary type of relationship depicted in the Chip folder is the hierarchy of modules of the design. The ISE inspects each design source file to see if it contains the definition of some module that has already been referenced in the design. If so, it attaches the new module under the referring module. If a newly added module references submodules that are already in the design, then

those positions under the new module are populated as well; otherwise, they are displayed as unfilled dependencies.

For single-chip designs there can only be one top-level design module under the Chip folder, much as there can only be one main() function in a C program. Under this top-level design module are to be found design constraints and system-level tests in addition to the lower level design submodules.

Design constraints typically contain additional information for processes that translate a design into a device configuration. Some constraints are applied directly, for example, configuration of external pins to an appropriate signaling standard. Others are less direct, for example, constraints on device fitters which must evaluate and balance many competing trade-offs for the speed, density, and power consumption of the fitted design. These can be given rules and cost functions in the constraints to guide them toward a design meeting overall goals.

System-level testing with simulation "test benches" is an important requirement in SOPC design. Given SOPC system complexity, it's a good idea to test at every stage, from individual module design, through system integration, to the fitted device. These test benches are usually written in hardware description languages (HDLs) such as Verilog [23] or VHDL [1] and provide descriptions of signal waveforms that drive the (simulated device) inputs, or monitors which observe and verify the outputs of a model of the (system or component) *unit under test* (UUT). The test benches will either be for *functional testing*, in which the logic of a module is exercised by waveform inputs operating in "unit time", or for *post-route testing*, where time-accurate models provided by the chip vendor for all of the actual components of the fitted design within the FPGA or CPLD, including gate delays, storage setup- and hold-time requirements, fan-out, and path delays, help in predicting whether the design will work in the real world.

4.2 ISE Processes window

The content of the Processes window changes according to which design component is highlighted in the Sources window. When a particular design element is highlighted, the Processes window will display in a tree structure all of the operations which the ISE can perform on that component. When these operations are compilation functions, they are organized in the order in which they must be performed, and marked with a small "spin" icon. For example, in Fig. 3, schematic Stage2Step2.sch ("Design top") is highlighted in the Sources window, hence the Processes window shows the first two major steps ("Translate" and "Map") required to implement the Stage2Step2 design. Within each major process step there may be a sequence of supporting processes that are necessary to complete the primary step—these are shown attached below the primary step in the tree structure and also marked with spin icons.

Other items attached in the Processes tree and marked with their own icons are reports (links to detailed reports on the results of the current process) and

optional services and utilities (which may be invoked to assist, improve, or analyze the operation of the related processing step). When the item in the process window is a link, selecting it will typically take the user out of the ISE window and into a specialized design environment for doing something such as creating constraints for the current component, generating schematic symbols of it, generating test fixtures for it, or any other capability the ISE provides for the component-type.

The current state of design compilation for the components processes is marked by icons next to each process, with <nothing> indicating that a process has not yet been run, "✔" indicating successful completion with no warnings, "**!**" indicating completion with warnings, "**X**" indicating failure to complete (errors) and "**?**" indicating "completed but stale" (design source has changed).

4.3 ISE Console window

The ISE console window provides narrative from the design processes as they operate. Most of the scripts that the ISE runs and messages that are emitted by the various processes can be seen as a running display in the Console window—watching it while a design is being processed can be quite informative. Occasionally, some message will flash by and trigger an insight into a design's characteristics or defects. When the processing of a design has finished or aborted, the "Warnings" or "Errors" tab will filter all of the process dialog down to the important details.

4.4 ISE Reports/Edit window

This provides a viewer for the various reports generated in the Processes window. It also becomes a very useful syntax-coloring editor for Verilog and VHDL sources in the Sources window.

5 SOPC Test and Sample Design

To illustrate the current state of SOPC design on FPGAs using widely available tools and devices we used the Xilinx Webpack tools to implement a small microprocessor system on a Xilinx 200K gate FPGA. A primary goal of the investigation was to verify the ability of the Xilinx tools to integrate non-proprietary IP components with different formats into a working whole. Other goals were to evaluate the ease of combining the results of the logic design tools with software design to achieve an integrated system development environment, and to test our ability to do system-level testing in the simulation environment.

The system to be constructed for the example consists of:

- MC6801 8-bit microprocessor;
- 2 kbyte program/data memory;
- inter-IC (I2C) serial communications controller;
- asynchronous communications interface adapter (ACIA);
- 1-bit input and output ports.

The target application connects a light-emitting diode (LED) to a switch using two microprocessor systems with a serial communications link between them. One system polls the switch and communicates the switch state to the other system, which turns the LED on or off accordingly. All components except switch and LED are implemented in a single FPGA.

5.1 Design tools for the SOPC test

The Xilinx ISE Webpack version 6.2i used for the test case is a recent (2004) version of Xilinx's free integrated software environment for CPLD and FPGA design downloaded from the Xilinx website [17] and installed on the test machine (Windows XP with 1.8 GHz CPU and 128 Mbytes of memory).

Webpack tools used in the project were as follows.

- ISE Project Navigator (PN): the main interface to the sources and tools for a design. PN presents the design's sources and their positions in the project structure, as well as their processing options. It also provides access to helper functions and resources.
- ISE Text Editor: a syntax-coloring text editor for Verilog, VHDL, and ABEL hardware description languages.
- Engineering Capture System (ECS): schematic editor for logic design. Linked to Xilinx component libraries and ISE processes.
- Xilinx Synthesis Technology (XST): compiles the FPGA programming configuration from design source files.
- logic simulator (Modelsim): provides logic simulation and results presentation for Webpack designs. The Modelsim XE simulator is not part of the Webpack but is obtained by a free download from the Model Technology division of Mentor Graphics (www.model.com). When installed it is tightly integrated with the PN to provide simulation for FPGA designs.
- BitGen: converts the compiled FPGA design into the *.bit* file used for programming the FPGA.
- iMPact: device programming interface; downloads the FPGA configuration to the FPGA.
- data2mem.exe: command-line utility provided in the Webpack for converting file types.

Additional tools available in the Webpack but not used for this project are:

- Constraints Editor: allows timing constraint specification;
- Pinout and Area Constraints Editor (PACE): allows hand-editing of FPGA pinout;

- State Machine Editor (StateCAD): allows specifying sequential logic as bubble diagrams;
- HDL Bencher: allows specifying test stimulus as waveforms;
- Floorplanner: allows hand-modification of component placement in the placed design;
- FPGA Editor: allows hand-modification of routing in the routed design.

Other tools used in the project:

- Dunfield ASM01 assembler: part of a package of cross-assemblers for various microprocessor targets. Found in xasm220.zip and available as freeware from various sites on the web. Dunfield Development Services [25], the originators, also have inexpensive updated versions of this product as well as the well-regarded Micro/C cross-compiler.
- hex2mem.exe: homebrew conversion routine for translating (assembler output) .hex files to *.mem* (Xilinx memory constraint) files.
- miscellaneous DOS command-line .bat scripts for converting assembler outputs into Xilinx FPGA configuration information.

5.2 Information resources for the SOPC test

A rich source of information concerning the Xilinx Webpack ISE is available at Xilinx's support website [15]. This provides an exhaustive look at and reference for the current ISE. Additionally, the ISE help menu provides access to hundreds of documents installed with the Webpack and hundreds more on Xilinx's website. Somewhat more accessible tutorials may be found both at Xilinx and at related vendors such as XESS Corp. (see, for example, [10]). Finally, the Webpack ISE itself has more than thirty built-in tutorial examples accessible from the file menu (File>Open Example...).

5.3 Design components for the SOPC test

All IP used in the system design except that from the Xilinx Spartan-IIE schematic symbol libraries was downloaded from www.opencores.org. The OpenCores group's mission statement is "to design and publish core designs under a license for hardware modeled on the Lesser General Public License (LGPL) for software", and it has become one of the best locations on the web for finding user-contributed designs. Design components for the project were selected on the basis that a) a Wishbone bus interface or buswrapper existed for them, and b) both Verilog and VHDL hardware description languages were represented in the mix of components.

Design components from opencores.org which were used in the project:

- cpu01.vhd/wb_cpu01.vhd: cpu01.vhd is a synthesizable VHDL model of the Motorola 6801 CPU written by John Kent. wb_cpu01 is the Wishbone buswrapper for it, written by Michael L. Hasenfratz, Sr.

- i2c_master_top.v: synthesizable Verilog model by Richard Herveille for the I2C (inter-IC) serial communications protocol promoted by Philips Electronics and others. This design builds the Wishbone bus directly into the top-level component in the design.
- miniuart.vhd/wb_acia.vhd: miniuart.vhd is a synthesizable VHDL model of a simple asynchronous serial communications interface written by Ovidiu Lupas. wb_acia.vhd is a Wishbone buswrapper for it, written by Michael L. Hasenfratz, Sr.

Design components from Xilinx which were used in the project:

- RAMb4_s8: 512x8 block RAM schematic symbol from the Xilinx "mem" library for the Spartan-II FPGA series. Four of these were used to construct the wb_4xRAMb4_s8.sch memory subsystem of the design.
- miscellaneous gates, decoders and multiplexers from the schematic symbol libraries for Spartan-II.

5.4 Target hardware for the SOPC test

The target hardware for the design was the D2SB FPGA prototyping board available from Digilent Inc. [12] for under $100. The D2SB board contains a 200K-gate Xilinx Spartan-IIE XC2S200E FPGA. All of the chip I/Os are brought out to 0.1-inch-center 40 pin headers around the board's periphery, and the FPGA can be programmed directly from the PC through a JTAG cable (∼$20 from Digilent), or indirectly through an (optional) on-board flash ROM. Several compatible accessory cards plug directly into the board's headers and add memory, digital or analog interface, and probing.

5.5 The SOPC design implementation sequence

The design was developed in several stages due to the uncertainties attached to the heterogeneous selection of components involved. The overarching philosophy was to first confirm the realizability of each component in the Xilinx FPGA, and then to integrate each component into the design in order of expected difficulty and/or importance to the final design.

Design realizability

Realizability is an important consideration for the use of untested components. Some problems which may arise are as follows.

- The HDL design may provide an accurate simulation of the desired element but may not be synthesizable, e.g., it may contain elements which a particular vendor's tools cannot translate into the FPGA configuration.

- Non-standard vendor-specific components that are not visible on initial inspection may be included in the design at a lower level. These components may not have direct correlates in the libraries for the target chip, a situation which can usually be worked around with a "roll-your-own" substitution, but at the cost of added design effort.
- The component may be too large for the intended FPGA target. This may be due to the component's natural complexity or to an inappropriate use of the target device's resources, for example, by casting memory into logic design when the target FPGA contains special-purpose memory arrays in addition to random logic.

For the first-stage test for component realizability we created separate designs for each of the components. As we do not require a practical result in terms of I/Os (as long as there are fewer I/Os than the chip permits), each component is made the top level of its individual design in the ISE, and the ISE "Map" and "Place & Route" processes are run. The .mrp (map report) and .par (place and route report) for each design gave the results shown in Table 1.

Table 1. Xilinx fitter reports

	wb_cpu01	i2c_master	wb_acia	tot.available
#errors	0	0	0	n.a.
#warnings	3	0	0	n.a.
#slice flipflops	206(4%)	132(2%)	91(1%)	4704(100%)
#LUTs	1411(29%)	237(5%)	148(3%)	2532(100%)

The results indicate that each of the components will compile in the Xilinx environment without many problems, and that chip size should easily be adequate for the design. Not including system memory, we should expect that roughly 10% (∼4%+2%+1%) of the chip's flip-flops and 40% (∼29%+5%+3%) of its logic resources (4-input LUTs) will be used by the system design.

5.6 System integration:

For the current design, many of the problems of system integration were assumed to have been solved at the component selection stage through the use of the Wishbone [22] standardized bus interface between components. Additionally, our desire was to maximize the efficient use of the Spartan-II FPGA's device resources through use of its special-purpose block RAM. Therefore, we needed to establish early in the design cycle whether we would be able to do the kind of program modification and testing that is important in the software development phase of an embedded system design. Hence, our next development priority was the test-integration of the Wishbone bus CPU with the

Spartan-II block RAM (BRAM). This step required the simultaneous development of our program-loading procedures, to prove that we could also meet our software development goals.

CPU/memory integration

We chose the Webpack's ECS schematic environment for the CPU-memory component integration over an HDL-based approach, as it was anticipated that the top level of the finished design would be in the form of a schematic, e.g., after proving the CPU-memory subsystem, the overall system design would be completed merely by adding the other components to it. Our preference for using schematics as the top level of the design stems in part from the use of multiple source languages in the project—the schematic provides a painless way to integrate different source types with the important bonus of allowing non-specialists to understand the overall system architecture without knowing the HDLs.

Our CPU-memory integration design is shown in Fig. 4, which was taken from the ECS schematic drawing for the design.

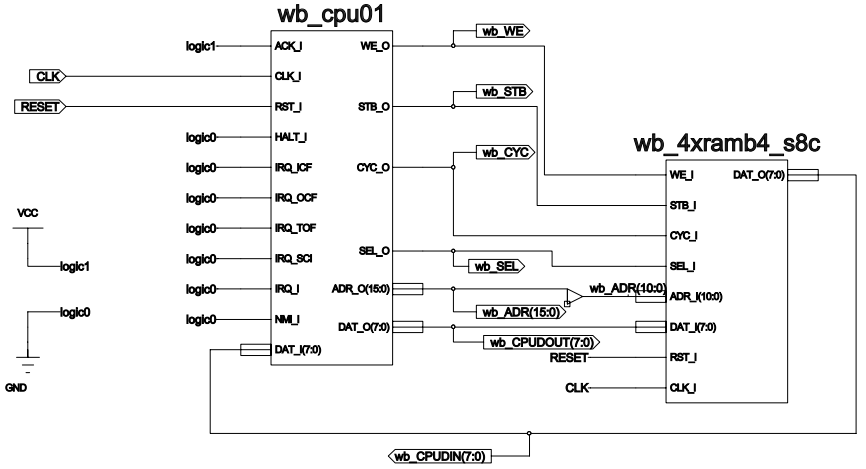


Fig. 4. SOPC CPU-memory design core

The device on the left in Fig. 4 is an automatically generated symbol for the VHDL-specified CPU design tested earlier. The Webpack ISE can generate schematic symbols for any component in the design hierarchy while maintaining links to the component's logic definition, thus providing a painless path to the incorporation of almost anything (including other schematics) into an overall design schematic. The ECS environment provides an editing facility

for such symbols which allows the designer to reorganize bus locations, text, or graphics on symbols to enhance their use in the design. This was used for symbols in the schematic to clean up connection paths of some of the major buses and control signals.

The memory device on the right in Fig. 4 is a symbol generated from a lower-level schematic "wb_4xramb4_s8.sch" containing four instances of the RAMB4_s8 512x8 byte BRAM component from the Xilinx component symbol library. Additionally, the lower-level memory schematic contains a minor amount of internal glue logic to translate the Wishbone bus interface to the RAMB4_s8 control interface, and to route the data bus signals in and out of the BRAM array.

Compilation of the above design was uneventful; however, the map report surprisingly showed a decrease in the amount of LUT logic required for the design. This caused some concern until it was noted that in the top-level design many of the CPU interrupt inputs were disabled, causing any circuits attached to them to be optimized out of the design, thus providing a net reduction in logic. (The gate count, which does not differentiate between gates used for logic and those used for memory, increases from around 10,000 for the CPU alone to around 65,000 for the CPU plus memory.)

Memory initialization

The next critical step of CPU-memory integration was to test our ability to install test software in the system memory. This proved to be the most difficult phase of the entire design sequence.

An essential fact of the Spartan-II FPGA chip architecture is that *all* of the storage elements of the FPGA (logic configuration, connection, and memory data) are loaded from an external source on startup. We like to take advantage of this fact by pre-placing the CPU's program data in the FPGA's BRAMs at startup. Some advantages of this approach are: 1) lower system parts count (versus a separate ROM memory for the microprocessor) and 2) software test early in the system development cycle, using the the hardware simulation environment.

Solving the BRAM startup configuration problem required understanding the meaning Xilinx attaches to the word "constraint" when used in the context of their FPGA design tools. Usually, constraints reflect instructions to the various compilation tools (mapping, placing, routing) and reflect the designer's requirements for speed, density, and I/O characteristics. Xilinx also takes constraints to mean the startup states of volatile storage elements in the design, as differentiated from the logic configuration and connection information stored in the FPGA's configuration array. There are a number of constraints which can be specified through the ISE menus, but alternative methods of constraint specification through input files are also provided. In particular, the BRAM's initializations can be specified through VHDL or Verilog source files when the design is described in one of those languages, or by

using Xilinx's proprietary *.ucf* (user constraint file) format for any source format, including schematics.

In theory, BRAM initialization constraints, including the compiled code for our microprocessor, can be edited into the .ucf file by hand to provide the system's software programming. However, we wanted to automate the conversion of software into FPGA programming files to allow normal code development methods to be used when the focus shifted from hardware to software development.

Xilinx provides a command-line utility, data2mem.exe, which can be used to generate the required .ucf files automatically. Unfortunately, the inputs which data2mem requires for this task (the *.bmm* file, a non-standard format containing a description of the BRAM memory instantiations in the design, and the *.mem* file, a non-standard format for representing binary data to be loaded into the BRAM) are not available from the standard set of tools in the ISE.

We solved the BRAM initialization problem by:

- constructing a .bmm file for the design "by hand." The .bmm file format [14] is obscure but straightforward, and the file only needs to be built once for a given hardware design.
- automating the translation of the *.hex* (Intel hex ASCII) output format of the assembler into the .mem format. To do this required that we write hex2mem.c, a simple (<100 lines including comments) program for the purpose.
- automating the loading of the above-produced .bmm and .mem files into the design via the data2mem.exe utility by writing several DOS scripts, asm2ucf.bat and asm2bin.bat. The asm2ucf script produced a .ucf file which was added to the design hierarchy in the ISE. There it became part of the instructions to the logic compiler, e.g., each time the code was changed and the design was recompiled, the updated assembly language binary of the microprocessor code was included in the logic specification of the BRAM. This provided the major benefit that even early simulations of the design demonstrated not only the design logic but also the software running on it. The asm2bin script uses the .bmm file and a generated .mem file through data2mem to edit an existing version of the *.bit* file containing a final FPGA programming configuration. The .bit file is the final output of the FPGA compilation processes and is usually sent directly to the FPGA or burned into a bootup memory device. However, Xilinx has also provided the capability, through the data2mem utility, for editing the contents of the BRAM entries (only) in the .bit file. Therefore, revisions of the onboard software stored in the BRAM can be made without disturbing the device logic design. Identifying this capability and using it will be our route to easy software development after hardware development is completed.

With our (presumed) solution of the software loading problem in hand, the next step was to run a test program in simulation on the CPU-memory subsystem design. To prepare for this we first used the Webpack ISE to automatically generate a Verilog "test fixture" for our design. The generated test fixture is a Verilog source file which conceptually sits "above" our design, e.g., it contains our entire design as a UUT instance, defines the set of signals into and out of the UUT, and initializes the values on the inputs to the UUT. Into this provided template we edited simple behaviors for the system RESET (active pulse at startup) and CLK (1 mHz oscillation), and then requested a simulation of the placed and routed design from the ISE.

Requesting a simulation in the Webpack ISE launches the Modelsim logic simulator, a separate product and interface. When the Webpack launches Modelsim it has prepared a great deal of information about the design—the view that Modelsim sees of the design is primarily taken from translations of the original design into the primitives (usually Verilog) from which the design realization will be constructed inside the FPGA. The Webpack sends a set of initial instructions to Modelsim in a *.ndo* script file for setting up the simulation displays and running the simulation. It then turns control over to the Modelsim console.

When the Modelsim console opens up it interprets the *.ndo* script and runs an initial test fixture simulation. If simulation is successful (not always the case, for a variety of reasons), then additional windows displaying the simulation's signals and waves pop up, and control is turned over to the user at the Modelsim console prompt.

The Modelsim simulator is an extremely complex, extremely powerful, and useful tool, worthy of an entire separate study. Good tutorials exist for it [20], but for our current purpose we merely need to use it as a logic analyzer for viewing program execution on our design. A wave view of the system signals is part of the initial default presentation. After editing the simulation presentation in the wave window (changing binary to hex bus displays, rearranging signal order) and saving the new format for future use in a wave.do file, we ran the simulation from the console for a few more microseconds of simulated time and then inspected the wave display to verify that the startup behaviors were as expected, i.e., that the RESET pulse caused the processor to read the reset vector from address FFFE, and that program execution then began at F900, the location of the reset label in the original source code. With this initial reassurance, we opened the list window, added and formatted the processor's bus signals, then set filters to limit the data displayed, in the same way we would choose clocks and qualifiers for using a logic analyzer in state mode. Fig. 5 shows fragments from a listing of the original source code, along with the associated wave and list displays from Modelsim's post-place and -route simulation of the system design.

Our conclusion, after examining a few dozen execution steps of the sample program, was that the CPU-memory subsystem is functional and, at least for the instructions tested, correct.

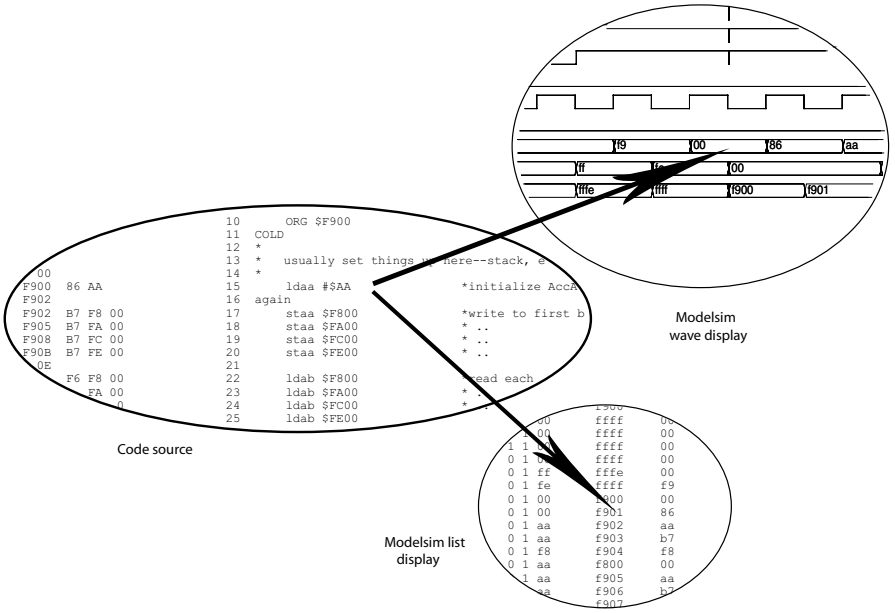


Fig. 5. CPU code execution on SOPC

Integrating the remaining components of the SOPC

The addition of the asynchronous serial port (ACIA) and I2C interface bus to the core CPU-memory design was straightforward due to the presence of the Wishbone bus interface on each component. A few additional decoders and multiplexers were required for selecting and routing data back to the CPU, but little more than connecting wires and buses were required, and the work was completed in less than an hour. The fitter map report indicated that about one-third of the device LUT logic was used for the single system design. Only a miniscule amount (∼6% of the device flip-flops) was used, which along with the four BRAM blocks gave an equivalent gate count of 77,000 for the entire design.

SOPC target application design

By creating a schematic symbol for the entire SOPC system and then placing two instances of the SOPC symbol connected by their serial ports in a new schematic, we provided the design for the target application (LED connected to switch by communicating microprocessor systems). Each copy of our SOPC in the new design was a complete standalone system, with one of them (CPU1) connected to the pushbutton input and the other (CPU2) to the LED. CPU1 monitors the pushbutton and sends its ASCII value ("0" or "1") over the serial link. CPU2 interprets the value received on the serial link and switches

the LED on or off accordingly. Pressing/releasing the pushbutton and seeing the LED turn on and off demonstrates the system operation.

SOPC application test

For those who lack access to sophisticated printed-circuit fabrication facilities, the Digilent D2SB FPGA prototyping board used for testing is an inexpensive way to plug an FPGA into a design; the board is little more than a platform to support the 208 pin plastic quad flat package (PQFP) FPGA and to bring its leads out to several rows of 0.1-inch socket headers. The Spartan-IIE on the Digilent board can be initialized either by downloading the programming *.bit* file using the programming cable from the PC or by programming an on-board flash ROM, also using the same cable but with a different configuration-jumper setting. We chose the former method (PC download) approach for quick results, and on downloading our application test, were rewarded by seeing the LED turn on and off when pressing/releasing the pushbutton.

6 After Action Report: Using the Xilinx Webpack for SOPC Design

The tools in the Webpack are almost unbelievably powerful, particularly in comparison to similar tools of only a few years ago. Through improvements in algorithms, increases in the computational power of desktop computers, and improvements in the chip architectures, the complexity of potential designs and the speed with which they can be realized have improved by orders of magnitude. However, there are sometimes mystifying failures of the ISE interface, the algorithms behind it, or the results it produces. Xilinx provides an excellent search engine [19] for its Answer Database—a compendium of questions and answers about issues experienced by users of the various Xilinx tools. The 20,000 or so entries in the Answer Database indicate Xilinx's seriousness in finding solutions to user's problems, but the number of problems stored there is probably just the tip of the iceberg in terms of actual difficulties in the field (for example, the thousands of users of the free Webpack cannot post their questions to Xilinx).

Over the course of several months' experience with the current version of the Webpack, certain recurring themes appeared regarding the types of problems which continually occurred, leading to an impression of fragility of the software. These were:

- path problems;
- hidden configuration values;
- failure reporting;
- SOPC software development.

6.1 Path problems

Path problems come in several flavors. One is that, even though SOPC designs will tend to accumulate components and complexity almost ad infinitum, the Webpack seems to prefer the "one big directory" form of design organization; that is, every tool expects to look in the common pool of files for its inputs and to leave its results there on completion of processing. Attempts to counter this tendency by creating component and version directories must be very carefully crafted and strictly followed to avoid problems of losing or using the wrong components. The Webpack itself attempts to address this issue in several ways, by providing archiving (all design components to a .zip file), creating libraries (one big file rather than one big directory), and "snapshots" (project "saveas" to a different directory), but none of the built-in approaches address issues of incremental design, portability, and reuse very well.

A second path problem seems to flow from the operating system heritage of various tools in the package—the hand-off between different tools often seems to involve (at least by inspecting error messages) conversions between UNIX, Windows network, and DOS conventions for file path descriptions, or even a mix of the various types, occasionally leading to some files hiding in plain sight: we can add them to the design, but the tools can't see them.

6.2 Configuration mysteries

To the credit of the Webpack's designers, almost every facet of the system behavior is controlled by a configuration value, and it seems that many configuration values are kept in ASCII text files which, if found, can be edited to modify the ISE behavior in a desired way. However, it also seems that some configurations, once set, have no path for being changed in an existing design (the design has to be restarted from scratch), some are capriciously changed as unremarked side effects of other operations, some are only conditionally effective and can be overruled by other (unidentified) settings, and some are only accessible via improbably unrelated menu paths or activity sequences. While information about how and where configurations are set can often be found in the Answer Database, the fact that the observed behavior was related to a configuration problem in the first place is usually obscure. Thus, solutions for this type of problem rely on good luck as much as good research.

6.3 Failure reporting

Occasionally, a process will fail with an error identification which leads directly to the solution in the Answer Database. Frequently, a process will fail while reporting something unrelated to the actual cause of failure, as is seen with software compiler error cascades. Once in a while, a process will report some problem but apparently does not fail; however, the expected output will also not be produced. Sometimes the exact diagnosis of the failure and its

recommended solution are reported, but the designer must look for the associated line in the hundreds of lines of reports and transcripts generated by one pass of the tools. On the plus side, and consistent with all of the other mysteries which surround us (Windows, Word, etc.), after a while the designer will develop a sixth sense about where things have gone wrong, and how to fix them (click here and type "Bob").

6.4 SOPC software development

The fact that we had to "roll our own" point tools to allow incorporation of software for the MC6801 processor used in the example into the hardware design and simulation cannot be held against Xilinx. This capability is present in their (non-free) Embedded Development Kit (EDK), which integrates software development functions into the ISE for their proprietary MicroBlaze soft-processor architecture. At a (much) higher level, Xilinx provides complete tools for development with the PowerPC 405 core embedded as "hard IP" (e.g., conventional custom logic) immersed in the fabric of their very large Virtex-II series of FPGA devices.

7 How To Learn More

The best introduction to SOPC design is to install one of the cited design environments ([3], [27], [7], [8], [13], [17]) or any of a number of others and embark on one's own SOPC design. The interested reader may also wish to investigate:

- an introductory text on Verilog and VHDL: Smith's *HDL Chip Design* [26];
- an in-depth VHDL text for design: Ashenden's *The Designer's Guide to VHDL* [1];
- an in-depth Verilog text for design: Palnitkar's *Verilog HDL, a Guide to Design and Synthesis* [23];
- a thorough introductory Webpack tutorial: XESS Corporation [10];
- the free-IP organization: Opencores.org [21]
- an on-line journal: *FPGA and Programmable Logic Journal* [28];
- a conference: Embedded Systems Conference [11].

References

1. P. J. Ashenden, *The Designer's Guide to VHDL, 2nd Edition*, Morgan Kaufmann Publishing, San Francisco, CA, 2002.
2. *Actel: Flash Devices: ProASIC PLUS*, Product information, Actel Corp., November 12, 2004, <`http://www.actel.com/products/proasicplus/index.html`>.

3. *Quartus II Software*, Product information, Altera Corp., September 27, 2004, <http://www.altera.com/products/software/products/quartus2/qts-index%.html>.
4. *Stratix II Devices: The Biggest & Fastest FPGAs*, Product information, Altera Corp., November 12, 2004, <http://www.altera.com/products/devices/stratix2/st2-index.jsp>.
5. *FPGA, CPLD and SERDES Programmable Logic Devices by Lattice Semiconductor*, Lattice Semiconductor Corp., November 12, 2004, <http://www.latticesemi.com/>.
6. *ispXPLD 5000MX Family*, Lattice Semiconductor Corp., May, 2004, <http://www.latticesemi.com/lit/docs/datasheets/cpld/5kmx.pdf>.
7. *Programmable Logic Software Development Tools by Lattice*, Lattice Semiconductor Corp., November 12, 2004, <http://www.latticesemi.com/products/devtools/software/index.cfm>.
8. *Mentor Graphics: The EDA Technology Leader*, Mentor Graphics Corp., November 12, 2004, <http://www.mentor.com/>.
9. *QuickLogic—Embedded Standard Products... Beyond Programmable Logic*, QuickLogic Corp., November 12, 2004, <http://www.quicklogic.com/>.
10. *Introduction to Webpack 6.1*, XESS Corp., October 30, 2003, <http://www.xess.com/appnotes/webpack-6_1-xsb.pdf>.
11. *The Embedded Systems Conferences*, Embedded.com, November 12, 2004, <http://www.esconline.com/>.
12. *Digilent D2-SB System Board Reference Manual*, Digilent Inc., September 18, 2003, <https://digilent.us/Data/Products/D2SB/D2SB-rm.pdf>.
13. *Synplicity: Products*, Synplicity Inc., November 3, 2004, <http://www.synplicity.com/products/index.html>.
14. *DATA2BRAM*, Xilinx Inc., April 11, 2003, <http://www.xilinx.com/ise/embedded/data2bram.pdf>.
15. *Software Manuals and Help—support.xilinx.com*, Xilinx Inc., November 12, 2004, <http://www.xilinx.com/support/sw_manuals/xilinx6/>.
16. *Spartan-IIE 1.8V FPGA Family: Complete Data Sheet*, Home page, July 9, 2003, <http://direct.xilinx.com/bvdocs/publications/ds077.pdf>.
17. *Xilinx: Design Tools Center*, Xilinx Inc., November 12, 2004, <http://www.xilinx.com/products/design_resources/design_tool/index.h%tm>.
18. *Xilinx: Programmable Logic Devices, FPGA and CPLD*, Xilinx Inc., November 12, 2004, <http://www.xilinx.com/>.
19. *Xilinx Support*, Xilinx Inc., November 12, 2004, <http://www.xilinx.com/support/mysupport.htm>.
20. *Modelsim Support*, Product support, Mentor Graphics Corp., November 12, 2004, <http://www.model.com/support/documentation.asp>.
21. *Opencores.org*, Opencores.org, November 12, 2004, <http://www.opencores.org/>.
22. *SoC Interconnection: Wishbone*, Opencores.org, November 12, 2004, <http://www.opencores.org/projects.cgi/web/wishbone/wishbone>.
23. S. Palnitkar, *Verilog HDL, A Guide to Design and Synthesis, 2nd Edition*, Prentice-Hall PTR, Indianapolis, IN, 2003.
24. *Xilinx tops ranks of FPGA and EDA vendors*, EEdesign.com, June 6, 2004, <http://www.eedesign.com/showArticle.jhtml?articleID=21401766>.
25. *Welcome to Dunfield Development Services*, Dunfield Development Services, June 29, 2004, <http://www.dunfield.com/>.

26. D. J. Smith, *HDL Chip Design: A Practical Guide for Designing, Synthesizing and Simulating ASICs and FPGAs Using VHDL or Verilog*, Doone Publications, Madison, AL, 1996.
27. *Cadence Design Systems*, Cadence Design Systems, November 12, 2004, <`http://www.cadence.com/`>.
28. *FPGA and Programmable Logic Journal*, Techfocus Media, Inc., November 12, 2004, <`http://www.fpgajournal.com/`>.

Part III

Software

Fundamentals of RTOS-Based Digital Controller Implementation

Qing Li

10501 Davison Avenue, Cupertino, CA 95014, U.S.A. qingli@speakeasy.net

1 Introduction

Digital controllers are increasingly designed and built using commercial off-the-shelf (COTS) hardware and software components. This is a result of the availability of high-performance, low-cost microcontrollers, specialized embedded processors, and various real-time operating systems (RTOSs). The wide selection of RTOSs does not necessarily translate into an increased availability of systems that are capable of facilitating the construction of controllers meeting the performance and stability requirements demanded by the mechatronics applications. A primary contributing factor to that disparity is that the design goals of many RTOSs focus on the overall system response time and may neglect other timing constraints in the controller.

Control engineers have made design assumptions about the capabilities of the RTOS that may not necessarily have been realized or supported by the actual software components. Control engineers often assume that the underlying software platform is fully capable of facilitating precisely time-triggered sampling of the controlled system, with deterministic, constant computation delay [27]. Some of the typical RTOS design goals are bounded context switch time, bounded response time to external interrupts, and preemptive priority-based scheduling, which result in good average system response time. These objectives point to an event-driven model that may be inadequate in addressing many of the timing attributes of the discrete-time model of the digital control system. The time variations in the control system due to sampling jitter, computation delay, and transient errors are not always accounted for by the RTOS.

Therefore, implementing control applications "on top" of an RTOS, which meet performance and stability requirements demands close collaboration between control engineering and computing science. There exist an overlapping problem space and complementary solutions between control engineering and the real-time computing branch of computer science. A digital controller implemented using an RTOS in an embedded processor can be classified as either

a *real-time system* or a real-time *embedded system*. The definition of a real-time system is a system in which overall correctness depends on both logical correctness and timing correctness [8], i.e., a real-time system produces the correct computation results in the defined time bound. A related attribute of real-time systems is predictability. An embedded system is defined as a system with tightly integrated hardware and software components, constructed to perform a dedicated function. Embedded systems are characterized by the choice of processor, level of application awareness, and the hardware and software codesign model used in system development. These concepts fit well in the control engineering domain.

The typical RTOS is comprised of a real-time kernel, a scheduler, the I/O subsystem, the memory subsystem, and application level programming support in terms of library functions. The kernel provides services, such as the timer services, and synchronization and communication primitives. The scheduler executes the scheduling algorithm for a task set. The I/O subsystem provides services to access the system I/O devices. The memory subsystem provides memory allocation, reclamation, and virtual-to-physical address mapping services. Each RTOS component has direct impact on the controller design and its performance; most notable are the scheduling algorithms and the synchronization primitives.

Fig. 1 illustrates a typical sampled-data feedback control system. Implementing this controller in an RTOS involves an iterative process. The first step is the decomposition of the controller components into corresponding RTOS tasks, for example, mapping the sampler into a periodic task. The next step in the implementation involves assigning task parameters such as period, execution time, and deadline to the various tasks. Then a scheduling algorithm is chosen for the task set followed by a schedulability analysis. Results obtained from the schedulability analysis can indicate that modifications may be necessary to the parameters of certain tasks. The schedulability analysis may also indicate that the controller decomposition is not optimal. Therefore, the decomposition is refined; adjustments are made to the parameters of tasks that still exist in the task set, followed by a subsequent schedulability analysis.

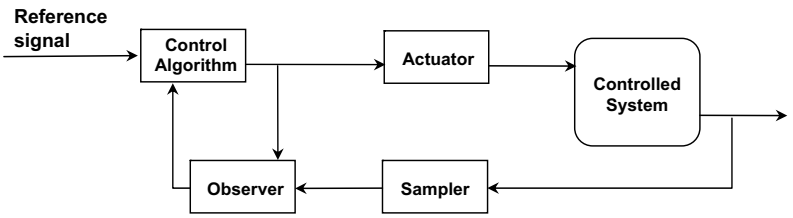


Fig. 1. Sampled-data control system block diagram

The choice of scheduling algorithm is crucial to the controller's performance. Computer scientists have made assumptions about the digital con-

troller design that have resulted in computing algorithms that produce suboptimal control performance in the end controller. The scheduling algorithms research community has assumed, as noted in [3], that a control algorithm can be modeled as a periodic task having a fixed period, that the control task has a hard deadline, and that its worst-case execution time (WCET) is known a priori. These assumptions do not apply to the types of controllers that are, for example, non-periodic, with alternating sampling periods and with a degree of tolerance for sampling delay and controller response time. The WCET of a task is difficult to obtain in general due to factors such as execution sequence dependencies and resource synchronization. Task synchronization and inter-task communication can cause variations in the time it takes to execute the control loop. The nature of synchronization, such as access to shared resources, can cause problems such as priority inversion, which is another source of sampling jitter. Priority inversion is described in Section 3.1. RTOS kernel context switch overhead can also trigger sampling jitter. The kernel context switch is described in Section 2.2. Scheduling overhead is a direct function of task partition. Therefore, control engineers can benefit greatly from detailed knowledge of the underlying RTOS by accounting for the characteristics of the RTOS in the controller design. For example, increasing the sampling rate up to a limit can increase the controller performance but may reduce task schedulability depending on the scheduling algorithm.

This chapter examines the major components of an RTOS and the implementation of the controller using those components from a computing science perspective. By introducing the various operating system concepts and the way in which these concepts affect the digital controller design, this chapter aims to aid control engineers in better understanding an RTOS. With this knowledge, design decisions can be made to compensate for the deficiencies in the underlying software platform. This chapter also provides a survey of the results of the various research topics in the field of computer-controlled systems design. The contribution of this chapter is its synthesis of topics from control engineering theory with concepts from real-time systems in the RTOS context. We thus aim to help the control engineer to develop a systematic approach to the implementation of control systems using either a COTS RTOS or an open source-based research RTOS.

The main objective of this chapter is to assist the control engineer in architecting the controller implementation while taking into consideration the issues that are present in the underlying supporting RTOS, thus creating an optimal design.

The remainder of this chapter is organized as follows. Section 2 defines the types of tasks, outlines task partition strategies, discusses the various scheduling algorithms, and introduces schedulability analysis. Section 3 introduces the common RTOS synchronization primitives, discusses the priority inversion problem, and describes the various solutions formulated for the priority inversion problem. Section 4 investigates the RTOS deficiencies that impact controller performance, stability, and reliability, and then discusses research

results that address those deficiencies. Section 5 enumerates several research RTOS that are useful for digital controller implementation.

2 Tasks, Scheduling Algorithms, and Schedulability Analysis

A task is a set of code that is scheduled and executed by the operating system. The RTOS kernel creates and maintains a data structure called the task control block (TCB) for each task. The TCB contains task-specific state information such as the address of the next instruction to execute, the stack pointer, a copy of the processor registers, and the I/O states of the task. A context switch refers to the kernel action of saving the execution state of one task into its TCB and replacing the processor states with the information from the TCB of another task.

During its execution a task competes with other concurrently executing tasks for system resources such as CPU time and system memory. Each scheduled invocation of a task is referred to as a *job* [2]. The time at which a job is ready for execution is referred to as its *release time*. A task is classified as a *periodic task* if the interval between job release times is constant. The sampler, shown in Fig. 1, is implemented as a periodic task. A task is classified as an *aperiodic task* if its release time is unpredictable. An event-driven task, such as a task created for handling device I/O, is an example of an aperiodic task because the interval between device interrupts is unknown. A task is classified as a *sporadic task* if its release time is unpredictable, but there is a guarantee of a minimum separation between jobs. A sporadic task can be considered as a special aperiodic task. In general, the minimum interval between the release times of successive jobs is called the *period* of the task.

2.1 Control application decomposition

Decomposing a control application into a task model involves a partition strategy. Partition by functionality implies mapping the *elementary functions* [27] of the control system into a task graph that identifies both the task set and the synchronization dependencies among these tasks. For example, the control system shown in Fig. 1 may include a task that samples the controlled system, a task that generates the reference signal, a task that perform the computation of the control algorithm, and an output task that communicates with the actuator, which results in the task graph [19] of Fig. 2.

In Fig. 2, each τ_i node represents a task, each rectangle denotes a resource, and each arrow represents a dependency relation between the adjacent nodes. For example, the task that performs the control-law computation depends on the data generated by the sampler task. This input data may also be modified by an operator. The output task that transmits the result to the actuator depends on the controller task to produce the output first.

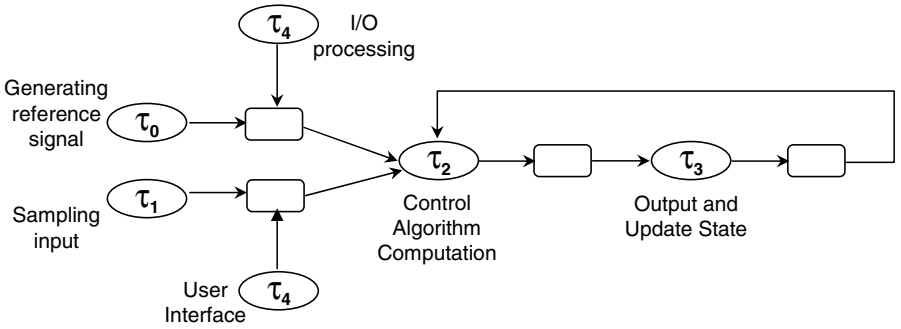


Fig. 2. Control system task graph

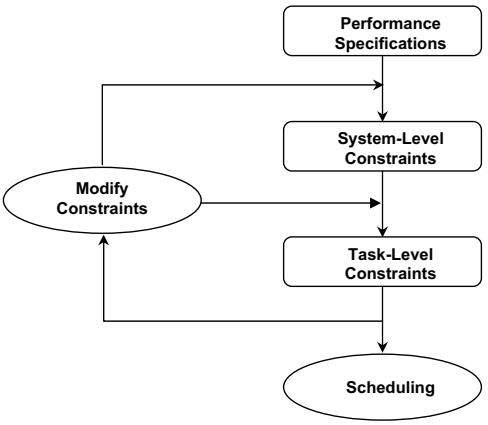


Fig. 3. Iterative design technique

Assigning periods and deadlines to the control tasks obtained from the application partition involves mapping the control system performance specifications into system-level timing constraints, translating the system-level timing constraints into task-level timing constraints (i.e., period and deadline), subsequently conducting task-level schedulability analysis, and then performing model refinement. Fig. 3 depicts this iterative derivation technique, which is described in [20].

The performance of a single-input single-output control system [19] can be characterized by the following features of its response to a step input: maximum overshoot, the rise time, the settling time in the transient state, and the steady-state error. The maximum overshoot M_{peak} is defined as the difference between the reference input and the maximum value of the response curve. The rise time T_{rise} is defined as the time required for the output response to reach a desired range with respect to the reference input. The settling time T_{set} is defined as the time required for the output response to reach a certain percentage, e.g., 95% of its steady-state value. The steady-state error E_{ss} is

defined as the limit as $t \to \infty$ of the difference between the reference input and the actual output.

The system-level timing constraints include sampling period and input-output latency. The mathematical model devised in [19] expresses M_{peak}, T_{rise}, T_{set}, and E_{ss} as functions of sampling period and latency. Solving these functions results in values for the period and latency that satisfy the system performance specifications. In this model, a continuous-time feedback controller is converted into a discrete-time system by introducing the zero-order hold (ZOH) block and a delay block between the controller and the plant. The delay block models the input-to-output delay that exists between the sensor and actuator. Using the z-transform method, M_{peak}, T_{rise}, T_{set}, and E_{ss} can be derived as increasing functions of period T^{Loop} and latency L^{Loop} from a closed-loop transfer function:

$$\begin{aligned} E_{ss} &= f_E(T^{loop}, L^{loop}) \\ M_{peak} &= f_M(T^{loop}, L^{loop}) \\ T_{set} &= f_s(T^{loop}, L^{loop}) \\ T_{rise} &= f_r(T^{loop}, L^{loop}). \end{aligned} \tag{1}$$

Applying the actual response specifications to the equations in (1) gives the inequalities in (2), where M'_{peak}, T'_{rise}, T'_{set}, and E'_{ss} denote the given constraints. The solutions to these inequalities are the values of T^{Loop} and L^{Loop} that meet the performance constraints.

$$\begin{aligned} f_E(T^{loop}, L^{loop}) &\leq E'_{ss} \\ f_M(T^{loop}, L^{loop}) &\leq M'_{peak} \\ f_s(T^{loop}, L^{loop}) &\leq T'_{set} \\ f_r(T^{loop}, L^{loop}) &\leq T'_{rise}. \end{aligned} \tag{2}$$

In a single-rate system, mapping the sampling period T^{Loop} and latency L^{Loop} into task periods and deadlines of a control loop is rather straightforward. However, the solution to deriving periods and deadlines of tasks of multiple control loops is both an optimization and an NP-complete problem. Assuming a multirate system that consists of $\{C_1, C_2, \ldots, C_n\}$ control loops, an algorithm is given in [19] for solving such a multirate system. The algorithm can be summarized as follows: for each control loop i, where $1 \leq i \leq n$,

1. Equate T_i^{Loop} and L_i^{Loop} to eliminate L from inequalities (2), and then solve (2) to obtain the maximum value for T_i^{Loop}.
2. Substitute the computed T back into (2) and then solve for L.
3. Once all of the T_i^{Loop} and L_i^{Loop} have been computed, derive the task periods and deadlines of the tasks from all of the control loops using the *Periodic Calibration Method* (PCM) [10].

4. If the PCM fails to find a period and deadline for each task such that the overall system satisfies the schedulability constraints, then identify the bottleneck control loop K, increase the T_k^{Loop} value, and go back to Step 2.

The above algorithm focuses on the problem space where the controller delay is less than the sampling period, which is the case in the majority of practical control systems. It is a much more complex problem to model a system where the delay is larger than the period. Also, this algorithm does not address the problem of timing variations that may be present in actual execution times.

Once the execution period and the deadline of a control task have been determined, it is then the responsibility of the RTOS scheduler to release a task for execution according to that period and to allow enough execution time for the task to complete before its deadline. A typical control system may include both sporadic and aperiodic tasks other than the periodic tasks. Therefore, the choice of the scheduling algorithm at the implementation phase determines the actual system schedulability in operation.

2.2 Scheduling algorithms

A *static scheduler* has the complete knowledge of the task set and the properties of each task for the lifetime of the system. The task properties include the task period, the deadline, the WCET, the resource requirements, and the execution dependency, which is also known as the precedence constraint. The arrival time of each task is also known a priori. Once the scheduler produces a feasible schedule, that schedule does not change during the lifetime of the running system.

A *dynamic scheduler* has complete knowledge of the task set and the attributes of these tasks at the time when a feasible schedule is produced. However, new task arrivals and the properties of these tasks are unknown to the scheduler. The dynamic scheduler makes a best effort to estimate the constraints of the newly arrived tasks at runtime, and to produce a new feasible schedule while continuing to satisfy the execution constraints of the existing task set. Therefore, the dynamic scheduler changes the schedule over time according to the current system state.

This scheduler classification [24] can be interpreted in the context of four scheduling paradigms summarized in [17]. These scheduling paradigms are static table-driven scheduling, static priority-driven preemptive scheduling, dynamic planning-based scheduling, and dynamic best-effort scheduling.

Static table-driven scheduling is most suitable for scheduling a set of periodic tasks. This scheduling approach produces a schedule table containing the task release times, finish times, and the order of execution, which takes into account the various constraints imposed on each task. The table is often derived from the WCETs of the tasks. Therefore, each task is guaranteed

to meet its timing, resource, and precedence constraints. A system that deploys the static table-driven scheduling paradigm has the highest degree of predictability, which is why such a scheduling paradigm is the most practiced approach in real-time control systems. A schedule table guarantees that every task can meet its deadline while satisfying its other constraints. On the other hand, the static table-driven scheduling scheme can produce a suboptimal schedule in the presence of non-periodic tasks.

Given a task set T with N periodic tasks, each task having a period T_i, the least common multiple (LCM) of periods of a task set is defined as the smallest value possible such that there exist N integers and the following equation is satisfied:

$$LCM = n_i T_i, \quad 1 \leq i \leq N. \tag{3}$$

In the context of the scheduling paradigm discussion, the value of the LCM represents a schedule length called the *scheduling horizon* in [18]. In (3), the value n_i represents the number of jobs of task i. In other words, the scheduler executes each job τ_i n_i times within each schedule period of the LCM, and the release times of these jobs and the order of execution are repeated every LCM units of time. Therefore, a feasible schedule derived for the LCM schedule period is a feasible schedule for the lifetime of the system.

Deriving a feasible schedule in the LCM scheduling horizon for a task set while satisfying the various types of dependencies and constraints is an NP-complete problem. The algorithms used for solving this problem have many attributes of the branch-and-bound class of search algorithms. The computation-intensive nature of the algorithms favors the off-line computation of the schedule, and this is one of the reasons why a subset of the static schedulers is classified as the *off-line* schedulers.

In preemptive priority-based scheduling methods, each task is assigned an importance level or priority, and at any point in time the executing task is the task with the highest priority among all existing tasks. If another task is created and has a higher priority, then the currently executing task is preempted, a context switch takes place, and the new task begins its execution. In a static, priority-driven preemptive scheme, a priority assignment algorithm specifies how priorities are assigned to tasks. The priority of a task, once assigned, does not change for the lifetime of the task. Schedulability analysis is performed offline, but no explicit schedule is produced. Instead, each task executes according to its priority at runtime.

Rate-monotonic scheduling is an example of the static priority-driven preemptive scheduling paradigm. Tasks are assigned priories using the rate-monotonic (RM) algorithm. The rate at which the jobs of a task are released is the inverse of the task period, i.e., the shorter the period, the higher the rate becomes. The task priority is directly proportional to its rate in the RM assignment scheme. For example, having two tasks A and B, the task with the higher rate will be assigned the higher priority of the two. It should be

clear from the context that task properties such as period and execution time must be known a priori in order to apply the RM scheduling algorithm.

The RM priority assignment policy will assign a low priority to a task that has a low job rate but a short deadline associated with urgent actions. Such a task will perform poorly with the basic RM algorithm. A variation of the RM policy called the *deadline-monotonic* (DM) assignment policy addresses this problem by assigning the task priority based on its deadline, i.e., higher priorities are assigned to tasks with shorter deadlines.

In the dynamic planning-based scheduling paradigm, the scheduler performs on-line schedule calculations when new tasks arrive in the system. The new schedule must guarantee that all those tasks from the old schedule can still meet their timing constraints, or else the new task is rejected by the system. An example algorithm of this category is given in [25].

In dynamic best-effort scheduling, a priority assignment algorithm assigns each task a priority. The scheduler selects the task with the highest priority for execution. The scheduler tries to satisfy each task's constraints; however, such a schedule is not guaranteed to exist. If that is the case, a schedule that maximizes the number of tasks that can run to completion while meeting their deadlines serves as the guideline for the scheduler. As such, the priority of a task may change during its lifetime, but each task is not guaranteed to meet all of its constraints. Many commercial RTOSs deploy the dynamic best-effort scheduling paradigm. Dynamic best-effort scheduling is the most flexible scheduling paradigm that accommodates the dynamics of the runtime system. However, this paradigm has the lowest degree of predictability. An example algorithm in this category is the Earliest Deadline First (EDF) algorithm. With EDF the task with the shortest remaining time to its deadline gets the highest priority. A newly arriving task can preempt an executing task if the newly arrived task is closer to its deadline and thus given a higher priority.

2.3 Handling sporadic and aperiodic tasks

A control system may contain non-periodic tasks to handle activities such as the device I/O and the operator interactions. The main objective in handling sporadic and aperiodic tasks is to have a fast response time without interfering with the periodic tasks. A common approach is to schedule non-periodic tasks during the idle times when there are no running periodic tasks. Another approach is to steal the slack time (also known as the *laxity*) of the periodic tasks when a non-periodic task arrives. If the deadline of the running periodic task is d, and the time at which the non-periodic task arrived is t, and the amount of time required to complete the job is x, then the slack time s is defined as $s = (d - t) - x$. Stealing the slack time may result in a better response time of the arriving aperiodic task.

One class of solutions models a special periodic task called a *server* to improve the aperiodic task response time. Examples of the various server designs are the *sporadic server*, the *deferrable server*, and the *bandwidth server*. The

server has a specific period and execution time and it is scheduled similarly to other periodic tasks. The execution time of the server is known as the *server capacity* or *budget* [4]. The server processes the aperiodic activities during its scheduled execution time slot. The *constant bandwidth server* (CBS) and the *total bandwidth server* (TBS) algorithms are discussed in Section 4 of this chapter.

2.4 Schedulability analysis

Schedulability analysis determines the feasibility of a schedule and evaluates the system timing behavior of a task set based on the characteristics of each task and the scheduling algorithm. Rate-monotonic analysis (RMA), developed by Liu and Layland in 1973, serves as the basis of hard real-time system analysis. There exists a set of standard assumptions about the system in order to apply RMA. These assumptions are given in [16] and are summarized in [8] as follows:

(a) The tasks in the system are all periodic.
(b) The tasks are independent of each other.
(c) The deadline of each task equals its period, i.e., $D = T$.
(d) Each task has constant execution time.
(e) Non-periodic tasks may exist but do not have hard deadlines.

The RMA with the above assumptions is called basic RMA. With basic RMA a set of tasks is schedulable if the following equation is satisfied:

$$U = \sum_{1}^{n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1). \tag{4}$$

The processor utilization factor U is defined as the fraction of time the processor spends to execute the entire task set. In (4), C_i is the WCET and T_i is the period of task i. The right-hand side of the inequality is called the theoretical upper bound, and it converges to 0.69 as the number of tasks increases, i.e.,

$$\lim_{n \to \infty} n(2^{1/n} - 1) = \ln 2 \approx 0.69. \tag{5}$$

[16] proves that a task set using the EDF algorithm is schedulable if the following inequality holds:

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \tag{6}$$

The assumptions made in basic RMA are too restrictive to be applied in practical real-time systems. As shown in Fig. 2, the tasks created for a control system will generally have some form of dependencies among them. These include communication, data, execution, and other types of synchronization dependencies. A low priority task may block a higher priority task in an RTOS

with preemptive priority-based scheduler. Therefore, the schedulability analysis needs to relax the second assumption and alter equation (4) by introducing the blocking factor:

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_i}{T_i} + \frac{B_i}{T_i} \leq i(2^{1/i} - 1), \quad 1 \leq i \leq n. \tag{7}$$

In (7), B_i is the worst-case blocking time that can be experienced by task τ_i. The fact that one task may block another task is the result of task synchronization.

The RM priority assignment is optimal when $D = T$ as stated in the third assumption for RMA. Instead, DM priority assignment is optimal when $D \leq T$. With DM priority assignment, the task with the shortest deadline has the highest priority. The response-time model is more appropriate for schedulability verification in this case. The worst-case response time of a task is given by an iterative equation [4],

$$R_i = C_i + \sum_{k \in hp(i)} \left\lceil \frac{R_i}{T_k} \right\rceil C_k, \tag{8}$$

where $hp(i)$ represents the tasks that have higher priorities than task i. In other words, the response time of task i is the sum of its computation time and the time of interference incurred on task i by tasks with higher priorities. Based on (8), a set of tasks is schedulable if for each task i, $R_i \leq D_i$.

3 Task Synchronization

Cooperating control tasks, as exemplified in Fig. 2, must synchronize their activities. Tasks that share data must synchronize their access to memory to ensure data integrity. This type of synchronization is referred to as *resource synchronization*. Related to resource synchronization are the concepts of *mutual exclusion* and *critical section*. As stated in Li [8]:

"Mutual exclusion is a provision by which only one task at a time can access the shared resource. A critical section is a section of code from which the shared resource is accessed. A mutual exclusion algorithm ensures that one task's execution of its critical section is not interrupted by the competing critical sections of other concurrently executing tasks."

The precedence constraint of a control task implies that the task must wait for one or more other tasks to complete before it can begin its execution. This type of synchronization is referred to as *sequence control* [8]. A task may need to transfer information to other tasks and wait for acknowledgement to arrive before resuming. This type of synchronization is referred to as *communication*.

A typical RTOS provides a set of synchronization primitives for the various synchronization needs in different applications. The implementation details of

these primitives directly impact the performance of the control application. Of interest to the discussion in this section are the resource synchronization primitives: semaphore, mutex, preemption lock, and interrupt lock.

3.1 Synchronization primitives and the priority inversion problem

A *semaphore* is a mutual exclusion primitive used to serialize access to shared resources. A task acquires the semaphore before it enters its critical section, and releases the semaphore when it exits the critical section. A semaphore has a number of tokens associated with it. A semaphore with a single token is called a *binary semaphore*. A semaphore with more than one token is called a *counting semaphore*. The operations that can be performed on a semaphore are acquire and release. A *mutex* is similar in nature to a binary semaphore but differs from the latter in that there is ownership information associated with a mutex. The owner of a mutex is the task that is currently holding that mutex. The operations that can be performed on a mutex can be lock and unlock. Only the current owner of a mutex can unlock that mutex. However, any task can release a semaphore (binary or counting) even when that task has not previously acquired the semaphore. As such, using binary semaphores for serializing access to shared data is more prone to programming errors.

Some RTOSs implement a primitive called the *preemption lock*. A preemption lock allows a task to disable the scheduler so that higher priority tasks cannot preempt it while it executes. The biggest issue with using a preemption lock for synchronization is that the preemption lock can disrupt a totally unrelated higher priority task from executing, thus rendering preemptive priority-based scheduling ineffective.

A task synchronizes access to shared data with an interrupt handler through the *interrupt lock*. Depending on the system hardware architecture, a task disables either the overall system interrupt or the device interrupt while accessing the shared data. The scheduler could possibly be affected by the manipulation of the system interrupt. In this case the interrupt lock has similar issue to those of the preemption lock. When tasks of varying priorities share common resources, situations arise in which a lower priority task can execute while a higher priority task is blocked and waiting to gain access to the shared resource. Such a situation is abnormal in an RTOS deploying a preemptive priority-based scheduling algorithm because there is a *priority inversion* in terms of order of execution. Priority inversions can be either bounded or unbounded. Timing anomalies as a result of an unbounded priority inversion must be avoided in a controller implementation.

Fig. 4 illustrates a *bounded priority inversion* situation. There exist two tasks τ_1 and τ_2 with priorities P_1 and P_2, respectively. Task τ_1 has a lower priority than task τ_2. τ_1 locks the mutex at time T_1. At time T_2 task τ_2 becomes ready to run and preempts τ_1 due to its higher priority, P_2. However, τ_2 is unable to access the shared resource because it cannot lock the mutex, which is held by τ_1. Therefore, τ_2 blocks at T_3 when τ_1 resumes execution. Priority

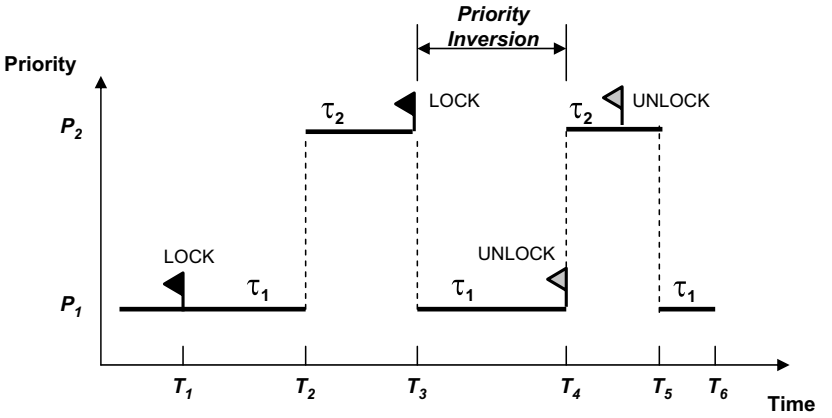


Fig. 4. Bounded priority inversion

inversion occurs at T_3. τ_1 unlocks the mutex at T_4 when it completes its access to the shared resource. This duration of the priority inversion is bounded by the access time of τ_1. The tasks τ_1 and τ_2 are said to have *competing critical sections*.

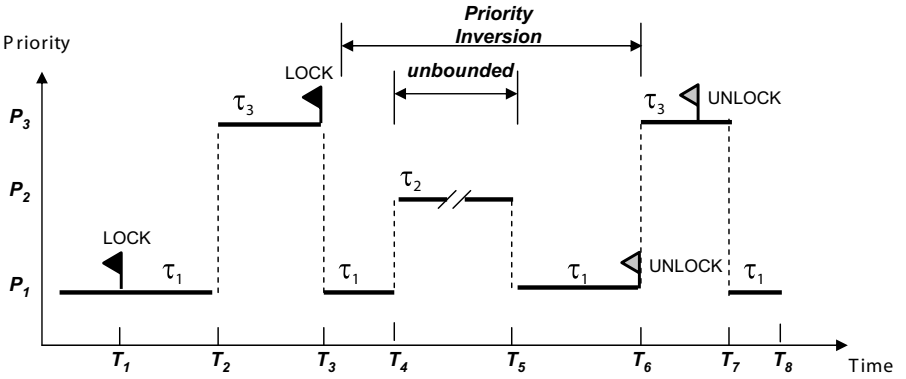


Fig. 5. Unbounded priority inversion

Fig. 5 illustrates a case of *unbounded priority inversion*. Task τ_2 has a higher priority than τ_1 but a lower priority than τ_3. Because τ_2 does not have a competing critical section, it preempts τ_1 and begins execution immediately at T_4 when τ_2 is ready to run. The interval between $[T_4, T_5]$ is unbounded because the execution time and runtime behavior of the unrelated task τ_2 cannot be determined.

Estimating the WCET of a controller task is nearly impossible when unbounded priority inversion can occur in a control system. Many controller

implementation algorithms break down without the WCET of the controller tasks. Implementing *resource access control protocols* as part of the synchronization primitive is one of the solutions to the unbounded priority inversion problem. Note that priority inversion cannot be eliminated altogether. However, the inversion time can be minimized.

3.2 Resource access control protocols

The well-known resource access control protocols are the *priority inheritance protocol* (PIP), *ceiling priority protocol* (CPP), and the *priority ceiling protocol* (PCP). Li [8] states:

"Resource access control protocol is a set of rules that define the conditions under which resource can be granted to a requesting task, and govern the execution scheduling property of the task holding the resource."

With the PIP when a task A of higher priority attempts to access a shared resource being held by a lower priority task B, the priority of B is raised to the priority of A temporarily until the time when B releases the resource. Referring to Fig. 5, the priority of task τ_1 is raised to P_3 at time T_3 until τ_1 completes its operation on the shared resource, at which time τ_1 assumes its previous priority P_1. The result is that at time T_4 the task τ_2 will not be able to preempt τ_1, thus allowing τ_1 to finish as soon as possible so that τ_3 can begin its execution. The PIP algorithm is a dynamic algorithm and has the transitive priority inheritance property, i.e., the priority of a task will be raised each time when a higher priority task arrives.

The CPP requires the priorities of all of the tasks to be known a priori. The resource requirements of each task are also known off-line, as is the highest priority task that will access each resource. This highest priority value is known as the *priority ceiling* of the resource. With CPP, each time a task gains access to a shared resource, its priority is temporarily raised to the priority ceiling of that resource. At any given time a task executes at the highest priority ceiling of all of the resources being held by that task. When a task releases a resource, its priority is lowered to the next highest priority ceiling, or the task resumes its original priority if it no longer holds any resource.

The PCP also requires advance knowledge of the priorities of all tasks and the resource requirements of each task. The *current priority ceiling* is the highest priority ceiling among all of the resources that are currently in use by different tasks. A requesting task τ_1 is granted the resource if that task has a higher priority than the current priority ceiling, or if the current priority ceiling belongs to a resource currently being held by that task. Otherwise, if the current priority ceiling belongs to a resource currently held by task τ_2, then τ_2 inherits the priority from τ_1 if τ_1's priority is higher. In general, the PCP guarantees that a task such as τ_1 will be blocked for at most one critical section. This property of the PCP is of special importance to controller implementation because each task can obtain a good estimate of its WCET.

Another property of the PCP is that no deadlock can ever occur in the running system, which is an essential attribute in safety-critical controller designs.

The resource access control protocols cannot be applied to the preemption lock and the interrupt lock mechanisms. Therefore, in situations where deploying such synchronization primitives is unavoidable, the period of deployment must be as short as possible.

4 RTOS Extensions for Control Engineering

The timing problems [28] that are present in control systems can be a combination of the control delays, jitter, and transient errors. Control delays include control-law computation time, and communication time if the control loop is closed over a network. Jitter includes sampling jitter, control computation start time jitter, and actuation jitter. Transient errors are the results of hardware errors and errors in the communication channel. Examples of transient errors include *vacant sampling* and *temporary blackout*. Task scheduling and synchronization can be direct sources of jitter. Another source of jitter is the arrival of non-periodic tasks. The majority of the well-known commercial and open source RTOSs are general-purpose RTOSs that evolve slowly and lack the solutions developed for control engineering. Many algorithms developed for the digital implementation of controllers can be found in research-grade RTOSs such as Asterix and S.Ha.R.K, or as extensions to existing RTOSs such as real-time (RT) Linux. These algorithms include task overrun management and jitter compensation through control loop decomposition, and these features are summarized next.

4.1 Task overrun management

The presence of aperiodic and sporadic tasks can overload a system, causing job overruns in periodic controller tasks. Section 2.3 discussed strategies of deploying a special type of periodic server task to handle aperiodic tasks. Job overruns may still exist depending on the controller design. The typical controller implementation assigns both fixed priorities and frequencies to the controller tasks. However, as controller designs grow more complex, such fixed frequency assignments based on WCETs produce suboptimal controller performance. Therefore, using the average execution time of the control loop to determine the task frequencies can achieve better resource utilization, leading to more optimized controller performance. The averaging approach can, however, cause a controller task to experience its WCET. Such a task is then considered an overrun task with respect to other controller tasks that are running within range of their average execution times. One solution [5] for task overrun management is to localize the overrun task by postponing its deadline and reducing its frequency to a lower bound necessary for guaranteeing the minimum acceptable controller performance. The formulation of the solution

in [5] is based on the work done by Seto et al. [21], which presents an algorithm that can compute and adjust the task frequencies according to the level of computing bandwidth utilization.

The fundamental principle behind the algorithm in [21] is that the higher the sampling frequency, the closer the digitized control-law implementation can approximate the continuous-time control algorithm. The corresponding task frequencies, however, must be chosen such that control performance optimization is subject to the overall system schedulability constraint. The implication is that the sampling frequency should be allowed to vary within a range, with the lower bound being the minimum frequency permissible in maintaining the system stability, and the upper bound being a multiple of the characteristic frequencies of the system. Since the control system performance is measured by the performance index, usually taken to be a function, J, of the sampling frequency, optimizing the system performance means minimizing

$$\Delta J(f) = J_D(f) - J^*, \tag{9}$$

where $\Delta J(f)$ is the difference between the PI of the continuous-time control $J_D(f)$ and its discrete-time implementation J^* at sampling frequency f. Approximating $\Delta J(f)$ by an exponential decay function, where α is the magnitude coefficient and β is the decay rate, gives

$$\Delta J(f) = \alpha e^{-\beta f}. \tag{10}$$

The control performance optimization problem then becomes [21]

$$\min_{(f_1,\ldots,f_n)} \Delta J = \sum_{i=1}^{n} w_i \Delta J_i = \sum_{i=1}^{n} w_i \alpha_i e^{-\beta_i f_i} \tag{11}$$

subject to the utilization constraint

$$\begin{aligned} &\sum_{i=1}^{n} C_i f_i \le A, \quad 0 < A \le 1 \\ &f_i \ge f_{mi}, \quad i = 1, \ldots, n, \end{aligned} \tag{12}$$

where f_i are the execution frequencies of the controller tasks. The term w_i is a weighting factor assigned to each task according to its relative importance in the task set. A is the available CPU bandwidth, f_{mi} is the minimum task frequency, and C_i is the computation time for a given task. Solving for the f_i in (11) yields [21]

$$\begin{aligned} f_i &= f_{mi}, \quad i = 1, \ldots, p \\ f_j &= \frac{1}{\beta_i}(\ln \Gamma_j - Q), \quad j = p+1, \ldots, n, \end{aligned} \tag{13}$$

where

$$\Gamma_j = \frac{w_j \alpha_j \beta_i}{C_j}, \quad Q = \frac{1}{\sum_{i=p+1}^{n} \frac{C_i}{\beta_i}} \left(\sum_{i=1}^{p} C_i f_{mi} + \sum_{i=p+1}^{n} \frac{C_i}{\beta_i} \ln \Gamma_i - A \right). \quad (14)$$

The value p is chosen such that

$$\sum_{i=1}^{p} C_i f_{mi} + \sum_{p+1}^{n} \frac{C_i}{\beta_i} \left(\beta_p f_{mp} + \ln \frac{\Gamma_i}{\Gamma_p} \right) \geq A. \quad (15)$$

This algorithm is applicable to the class of control systems where each $\Delta J(f)$ is a monotonically decreasing convex function. The convexity property guarantees a unique solution for the optimization problem.

The condition $A < 1$ is true when some task frequencies are close to their minimums. This condition holds when the digital controller implementation includes non-control-related tasks. Allowing the value of A in (12) to vary implies the algorithm can compute optimal task frequencies in accordance with the changing bandwidth conditions at runtime. One of the sources for variations in A can be the arrival of aperiodic tasks. The computation time C_i is the WCET of a given task, which means that the system typically performs at a suboptimal level. A better solution would be to substitute the normal execution time C_i^n into the constraint equation (12) and obtain an optimal frequency for the normal execution state. Task overrun management is then necessary to ensure that task deadlines are met with the optimal scheduling frequency f_i^{opt} obtained from (13). The task overrun management algorithm presented in [5] assigns each task a processor bandwidth,

$$U_i = f_i^{opt} c_i^n. \quad (16)$$

The TBS algorithm [23] assigns a deadline to the aperiodic request when it arrives at the server task,

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}, \quad (17)$$

where r_k is the kth aperiodic job arrival time, d_k is the deadline, d_{k-1} is the deadline of the $(k-1)$-th job, C_k is the computation time of job k, and U_s is the server bandwidth. Adopting this deadline assignment model in the periodic task overrun management algorithm results in

$$d_k = \max(r_k, d_{k-1}) + \frac{1}{f_i^{opt}}. \quad (18)$$

This deadline d_k is postponed by a factor of $\frac{WCET_i - C_i^n}{U_i}$ when a task exceeds its normal execution time C_i^n. This postponement assumes a maximum overrun due to the use of WCET in the calculation. The algorithm represented by (13) can ensure that each task is scheduled at a frequency that is at least

f_{mi}; however, missing one or more job deadlines will degrade the system performance. Therefore, the lower bound frequency must be modified to account for the overrun, thus giving a new minimum permitted frequency f'_{mi} given by

$$f'_{mi} = f_{mi} \frac{WCET_i - C_i^n}{U_i}. \tag{19}$$

The task set can be scheduled using the EDF algorithm and executes according to the CBS algorithm [1], which can provide better response time for the overrun task. In the original CBS algorithm, a server is assigned to handle each task. When a task overruns its allocated bandwidth, its deadline is postponed by an amount equal to the server period, and the server is replenished with its maximum allocated bandwidth. Such a new deadline assignment scheme for the overrun task can exceed the hard deadline of a periodic control task. In the modified CBS algorithm, called the CBS^{hd} algorithm, the server is replenished with the remaining execution time C_i^r if the remaining execution time is less than the maximum allocated bandwidth. In addition, the new deadline of the overrun task is postponed by $\frac{C_i^r}{U_i}$, which results in a deadline that will not exceed the hard deadline threshold.

4.2 Jitter compensation

As shown in Fig. 2, one method of reducing jitter is to decompose a controller into several essential entities: the data collection task, the control-law calculation task, and the output and update state task. The latter is responsible for transmitting the computation output to the actuator and updating the system state for the next sampling interval. The motivation behind such decomposition is to alleviate the problems encountered in the single-task controller implementation. The typical control loop can be modeled as a sequence of functions [6]:

```
LOOP
        WaitFor(Periodic-Clock-Interrupt)
        A/D Conversion
        CalculateOutput
        D/A Conversion
        UpdateState
END
```

In a single-task implementation, one task priority assignment determines the degree of all types of jitter. If the priority is not assigned sufficiently high, then other system tasks, both related and unrelated, with higher priorities can preempt the control loop in the midst of sampling or can prevent it from being scheduled in time; both situations result in sampling jitter. Similarly, preemption can occur during the CalculateOutput function, causing computation delay. As can be seen, in a single-task implementation, the occurrence

of jitter or delay in any function can trigger cascading jitter and delay effects throughout the control loop. On the other hand, if the control task is given a high priority to accommodate the sampling function, such a priority may cause a computation delay in control tasks belonging to other control loops.

Therefore, decomposing the control loop into subtasks facilitates the scheduling of elementary functions according to different priorities and deadlines. The A/D and D/A conversions are assumed to be negligible and are excluded from the scheduling analysis. The data collection task is given a high priority and has a short execution period in which limited workload is performed. The task has a small context to reduce the context switch overhead. Such a high priority but low cost task ensures that fixed interval periodic sampling has negligible jitter. The last two tasks are referred to as the CalculateOutput (τ_{CO}) task and the UpdateState (τ_{US}) task, respectively, in [6]. These two tasks have different deadlines. In general, the τ_{CO} task should complete as early as possible, while the τ_{US} task must complete before the start of the next sampling interval. The constraints on the deadlines can be expressed as [6]

$$\begin{aligned} D_{US_i} &= T_i \\ C_{CO_i} \leq D_{CO_i} &\leq T_i - C_{US_i}, \end{aligned} \tag{20}$$

where C_{CO_i} and C_{US_i} are the computation times of the subtasks τ_{CO} and τ_{US}, respectively, and T_i is the period of the overall control loop. The DM priority assignment algorithm is the optimal method for assigning task priorities once the deadlines D_{CO_i}, D_{US_i} are determined. Since the τ_{CO} is the more critical task, Cervin [6] presents heuristics in finding the optimal deadlines for the τ_{CO} subtasks while maintaining schedulability:

1. Let $D_{CO_i} = T_i - C_{US_i}$.
2. Assign DM priorities, i.e., shorter deadlines map to higher priorities.
3. Calculate the task response time R_i from (8).
4. Set $D_{CO_i} = R_i$.
5. Go back to Step 2 until no more improvements can be made to D_{CO_i}.

5 Survey of Control-Oriented RTOS

The characteristics of a typical RTOS stated by Stankovic and Ramamritham in 1991 [25] are still largely true today in the majority of the RTOSs, both commercial and open source. These characteristics include

- multitasking with preemptive priority scheduling
- efficient context switching
- efficient interrupt service facility
- flat memory model without support for virtual memory
- synchronization and communication primitives

- application level timing services
- small memory footprint.

An RTOS with the above characteristics is event driven, ensures fair sharing of the system resources and CPU time, and has little or no knowledge of the nature of any task that executes in it. The event-driven model measures the success and the applicability of the RTOS by the overall average system response time. The previous sections have shown that the response-time measurement neglects many timing constraints found in control applications, e.g., the deadline constraint of a control task. Commercial and open source general-purpose RTOSs may evolve slowly; however, results from the state-of-the-art research in RTOS design are visible in many research RTOSs, most notable are the Spring kernel [25], the Asterix real-time kernel [7], [13], the S.Ha.R.K RTOS [9], [22], and the Fiasco μ-kernel for Linux [14], [15]. These research kernels may have little commercial support but do include features such as non-blocking synchronization methodology, various scheduling algorithms, aperiodic servers, and new resource access control protocols. A research-grade RTOS can be considered as a control-oriented or control-enabling RTOS.

An RTOS supporting control applications must have a facility that allows a control application to specify the deadline of each task and the *hardness* of each deadline. The hardness of the deadline is not determined by the tightness of the deadline, but is determined by the level of tolerance in the system in dealing with the missed deadline. Consequently, an RTOS must have the ability to classify a task and account for that classification in the scheduler. For example, the Spring kernel classifies tasks into *critical*, *essential*, and *non-essential tasks*; the Asterix kernel classifies tasks into *hard periodic*, *soft periodic*, *hard aperiodic*, and *soft aperiodic tasks*. The S.Ha.R.K kernel classifies a task according to its criticality, which can be one of *hard*, *soft*, *firm*, and *non-real-time*. A control-enabling RTOS maintains other task parameters such as periodicity and resource requirements in the TCB. For example, the Asterix kernel maintains task period, deadline, best-case execution time (BCET), and WCET.

The previous sections have shown that many of the algorithms presented should be implemented as part of the underlying RTOS. Any rate adaptation solution requires the RTOS to support the measurements of actual task execution times. Online statistics-gathering and feedback are necessary for implementing the task overrun management algorithm. Such features are typically missing from commercial RTOSs. On the other hand, the Asterix kernel measures the execution time of each task and facilitates the implementation of deterministic replay for debugging real-time applications [26].

A typical commercial RTOS may implement the PIP but may lack any support for the PCP. The S.Ha.R.K RTOS comes with modules that implement the PIP, the PCP, and the *Stack Resource Policy*. The Spring kernel allows a task to make resource reservations. Once granted, the task is guaranteed to have exclusive access to reserved resources for its scheduled interval. The

non-blocking synchronization algorithms, e.g., the *wait-free locking* with helping [12], [11] are implemented in Asterix, S.Ha.R.K, and Fiasco. The Asterix kernel implements an *immediate inheritance protocol* [7] for resource access control. It also implements the wait and lock-free channel for inter-task communications. In addition, the Asterix kernel minimizes execution time jitter using dummy paths and dummy code.

These research kernels support the various priority assignment and scheduling algorithms such as EDF and RM scheduling. The S.Ha.R.K kernel allows pluggable modules for scheduling aperiodic tasks. These modules implement the different types of servers, e.g., the Sporadic Server, the CBS, and the TBS. The Spring kernel implements *admission control*, which determines whether a new task can be created and scheduled without jeopardizing any existing critical tasks.

These RTOSs are designed with control applications in mind and are freely available. The attractive attributes of these systems are that a researcher can either develop new algorithms, or improve upon the works done by others, and immediately put these works into action. The participants of these RTOS project communities have similar research interests; therefore, assistance and valuable insights are always readily available. The trade-offs of using a public open source research RTOS for digital controller implementation, however, are the limited support of hardware platforms, the limited development and debug tools, and the limited runtime analysis tools. For example, the non-blocking synchronization primitives found in the research RTOSs rely on the processor to have the atomic *compare-and-swap* instruction. This type of instruction is missing from many processor architectures such as the Intel x86 processor family. A broader issue is the lack of *board support packages* (BSPs) that typically provide a suite of drivers and system software for a wide variety of hardware devices and processors. A commercial RTOS such as VxWorks fills those gaps in the BSP and tools areas. A considerable amount of time may be spent in producing working bootstrapping code. Having a commercially available *on-chip debugging* (OCD) solution can dramatically reduce that development effort.

6 Concluding Remarks

The research field for digital controller implementation is vast and diverse. This chapter has attempted to synthesize the relevant results of various research works from both control engineering and RTOS designs into a systematic approach applicable in RTOS-based control design and implementation. The challenge was in selecting the topics and presenting the information in a succinct fashion. The main focus of this chapter was on single-processor and single-rate digital controller architecture. As such, topics in controller implementation on multiprocessor architectures and distributed controller design

where the loop is closed over a network have been mostly excluded from the discussions.

References

1. L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proc. IEEE Real-Time Systems Symposium*, Dec. 1998.
2. J. H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Transactions on Computer Systems*, 15(2):134–165, May 1997.
3. K-E. Årzen, B. Bernhardsson, J. Eker, A. Cervin, K. Nilsson, P. Persson, and L. Sha. Integrated control and scheduling. Technical Report ISSN 0820-5316, Dept. Automatic Control, Lund Institute of Technology, Sweden, 1999.
4. G. Buttazzo. Real-time operating systems: Problems and novel solutions. In *Proc. of 7th Int. Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems*, Oldenburg, Germany, Sept. 2002.
5. M. Caccamo, G. Buttazzo, and L. Sha. Handling execution overruns in hard real-time control systems. *IEEE Transactions on Computers*, 51(7), 2002.
6. A. Cervin. Improved scheduling of control tasks. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 4–10. York, U.K., June 1999.
7. H. Thane, A. Pettersson, and D. Sundmark. The Asterix real-time kernel. In *Proceedings of the 13th Euromicro International Conference On Real-Time Systems*, June 2001.
8. Q. Li and C. Yao. *Real-Time Concepts for Embedded Systems*. CMP Books, 2003.
9. P. Gai, L. Abeni, M. Giorgi, and G. Buttazzo. A new kernel approach for modular real-time systems development. In *Proceedings of the 13th IEEE Euromicro Conference on Real-Time Systems*, Delft, Netherlands, June 2001.
10. R. Gerber, S. Hong, and M. Saksena. Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Transactions on Software Engineering*, 21(7), July 1995.
11. M. Greenwald and D. Cheriton. The synergy between non-blocking synchronization and operating system structure. In *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), Seattle WA*, pages 123–136. USENIX, Oct. 1996.
12. M. Hohmuth and H. Hartig. Pragmatic non-blocking synchronization for real-time systems. In *Proc. of the USENIX Annual Technical Conference*, 2001.
13. Asterix kernel. http://www.mrtc.mdh.se/projects/asterix/.
14. Fiasco kernel. http://os.inf.tu-dresden.de/fiasco/.
15. L^4 Linux. http://os.inf.tu-dresden.de/l4/linuxonl4/.
16. R. Liu and Layland. Scheduling algorithms for multiprogramming in hard-real time environment. *Journal of the Association of Computing Machinery 20*, 20(1):174–189, 1973.
17. K. Ramamritahm and J. A. Stankovic. Scheduling algorithms and operating systems support for real-time systems. *Proceedings of the IEEE*, pages 55–67, Jan. 1994.

18. O. Redell. Global scheduling in distributed real-time computer systems-An automatic control perspective. Technical Report TRITA-MMK 1998:6, ISSN 1400-1179, ISRN KTH/MMK-98/6-SE, Department of Machine Design, Royal Institute of Technology, S-100 44 Stockhom, 1998.
19. M. Ryu and S. Hong. Toward automatic synthesis of schedulable real-time systems: From the temporal perspectives. *International Journal of Integrated Computer-Aided Engineering*, 5(3):261–277, May 1998.
20. M. Saksena. Real-time system design: A temporal perspective. In *Proc. of IEEE Conference on Electrical and Computer Engineering*, pages 405–408, Waterloo, Canada, May 1998.
21. D. Seto, J.P. Lehoczky, L. Sha, and K.G. Shin. On task schedulability in real-time control system. In *Proc. IEEE Real-Time Systems Symposium*, Dec. 1996.
22. S.Ha.R.K. http://shark.sssup.it.
23. M. Spuri and G. Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *The Journal of Real-Time Systems*, 10(2):179–210, March 1996.
24. J. Stankovic, M. Spuri, M. Di Natale, and G. Buttazzo. Implications of classical scheduling results for real-time systems. *IEEE Computer*, pages 16–25, June 1995.
25. J. A. Stankovic and K. Ramamritham. The Spring kernel: A new paradigm for hard real-time operating systems. *IEEE Software*, 8(3):62–72, May 1991.
26. H. Thane and H. Hansson. Using deterministic replay for debugging of distributed real-time systems. In *Proceedings of the 12th Euromicro Conference on Real-Time Systems*, pages 265–272, June 2000.
27. M. Törngren. Fundamentals of implementing realtime control applications in distributed computer systems. *J. of Real-Time Systems*, 14:219–260, 1998.
28. B. Wittenmark, J. Nilsson, and M. Törngren. Timing problems in real-time control systems: Problem formulation. In *Proc. of the American Control Conference*, 1995.

Implementation-Aware Embedded Control Systems

Karl-Erik Årzén, Anton Cervin, and Dan Henriksson

Department of Automatic Control, Lund University,
Box 118, SE-221 00 Lund, Sweden
{karlerik,anton,dan}@control.lth.se

1 Introduction

The current pervasive and ubiquitous computing trend has increased the emphasis on embedded and networked computing within the engineering community. Today embedded computers already by far outnumber desktop computers. Embedded systems are often found in consumer products and are therefore subject to hard economic constraints. Some examples are automotive systems and mobile phones. The pervasive nature of these systems generates further constraints on physical size and power consumption. These product-level constraints give rise to resource constraints on the computing platform level, for example, limitations on computing speed, memory size, and communication bandwidth. Due to economic considerations, this is true in spite of the fast development of computing hardware. In many cases, it is not economically justified to add an additional CPU or to use a processor with more capacity than what is required by the application. Cost also favors general-purpose computing components over specially designed hardware and software solutions.

Control systems constitute an important subclass of embedded computing systems—so important that, for example, within automotive systems, computers commonly go under the name electronic control units (ECUs). A top-level modern car contains more than 50 ECUs of varying complexity. Most of these ECUs implement different feedback control tasks, for instance, engine control, traction control, anti-lock braking, active stability control, cruise control, and climate control.

At the same time, control systems are becoming increasingly complex from both the control and computer science perspectives. Today, even seemingly simple embedded control systems often contain a multi-tasking real-time kernel and support networking. Many computer-controlled systems are distributed, consisting of computer nodes and a communication network connecting the various systems. It is not uncommon for the sensor, actuator, and control calculations to reside on different nodes. This gives rise to networked

control loops. Within individual nodes, controllers are often implemented as one or several tasks on a microprocessor with a real-time operating system. Often, the microprocessor also contains tasks for other functions, such as communication and user interfaces. The operating system typically uses multiprogramming to multiplex the execution of the various tasks. The CPU time and the communication bandwidth can hence be viewed as shared resources for which the tasks compete.

By tradition, the design of computer-based control systems is based on the principle of *separation of concerns*. This separation is based on the assumption that feedback controllers can be modeled and implemented as periodic tasks that have a fixed period, T, a known worst-case bound on the execution time (WCET), C, and a *hard deadline*, D. The latter implies that it is imperative that the tasks always meet their deadlines, i.e., that the actual execution time (response time) is always less or equal to the deadline, for each invocation of the task. This is in contrast to a *soft deadline*, which may occasionally be violated. The fixed-period assumption of the simple task model has also been widely adopted by the control community and has resulted in the development of the sampled computer-control theory with its assumption of deterministic, equidistant sampling. The separation of concerns has allowed the control community to focus on the pure control design without having to worry about how the control system eventually is implemented. At the same time, it has allowed the real-time computing community to focus on development of scheduling theory and computational models that guarantee that hard deadlines are met, without any need to understand what impact scheduling has on the stability and performance of the plant under control.

Historically, the separated development of control and scheduling theories for computer-based control systems has produced many useful results and served its purpose well. However, the separation has also had negative effects. The two communities have become partially alienated. This has led to a lack of mutual understanding between the fields. The assumptions of the simple model are also overly restrictive with respect to the characteristics of many control loops. Many control loops are not periodic, or they may switch between a number of different fixed sampling periods. Control loop deadlines are not always hard. On the contrary, many controllers are quite robust to variations in sampling period and response time. Hence, it is questionable whether it is necessary to model them as hard-deadline tasks.

The main drawbacks of the separation of concerns are that it does not always utilize the available computing resources in an optimal way, and that it sometimes gives rise to worse control performance than what can be achieved if the designs of the control and real-time computing parts are integrated. This is particularly important for embedded control applications with limited computing and communication resources, demanding performance specifications, and high requirements on flexibility. For these types of applications, better performance can be achieved if a codesign approach is adopted, where the control system is designed taking the resource constraints into account

and where the real-time computing and scheduling is designed with the control performance in mind. The resulting *implementation-aware control systems* are better suited to meet the requirements of embedded and networked applications.

2 The Codesign Problem

The *control and scheduling codesign problem* can be informally stated as follows in the uniprocessor case:

> Given a set of processes to be controlled and a computer with limited computational resources, design a set of controllers and schedule them as real-time tasks such that the overall control performance is optimized.

An alternative view of the same problem is to say that we should design and schedule a set of controllers such that the least expensive implementation platform can be used while still meeting the performance specifications. For distributed systems, the scheduling is extended to also include the network communication.

The nature and the degree of difficulty of the codesign problem for a given system depend on a number of factors:

- *The real-time operating system.* What scheduling algorithms are supported? How is I/O handled? Can the real-time kernel measure task execution times and detect execution overruns and missed deadlines?
- *The scheduling algorithm.* Is it time driven or event driven, priority driven or deadline driven? What analytical results regarding schedulability and response times are available? What scheduling parameters can be changed on-line? How are task overruns handled?
- *The controller synthesis method.* What design criteria are used? Are the controllers designed in the continuous-time domain and then discretized or is direct discrete design used? Are the controllers designed to be robust against timing variations? Should they actively compensate for timing variations?
- *The execution-time characteristics of the control algorithms.* Do the algorithms have predictable worst-case execution times? Are there large variations in execution time from sample to sample? Do the controllers switch between different internal modes with different execution-time profiles?
- *Off-line or on-line optimization.* What information is available for the off-line design and how accurate is it? What can be measured on-line? Should the system be able to handle the arrival of new tasks? Should the system be re-optimized when the workload changes? Should there be feedback from the control performance to the scheduling algorithm?

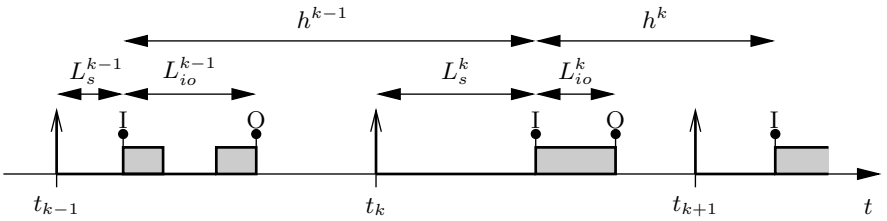


Fig. 1. Controller timing

- *The network communication.* Which type of network protocol is used? Can the protocol provide worst-case guarantees on the network latency? How large is the probability of lost packets?

Codesign of control and computing systems is not a new topic. Control applications were one of the major driving forces in the early development of computers. At that time, limited computer resources was a general problem, not only a problem for embedded controllers. For example, the issues of limited word length and fixed-point calculations and their results on resolution were well known among control engineers in the 1970s. However, as computing power has increased, these issues have received decreasing attention. A nice survey of the area from the mid-1980s is given in [14].

3 Temporal Determinism

Computer-based control theory normally assumes equidistant sampling and negligible, or constant, input-output latencies. However, this situation can seldom be achieved in practice or is too costly for a particular application. In a multi-threaded system, tasks interfere with each other due to preemption and blocking from task communication. Execution times may be data dependent or vary due to the use of caches. In networked control loops, where the sensors, controllers, and actuators reside on different physical nodes, the communication gives rise to latencies that can be more or less deterministic, depending on the network protocols used. The result of all this is jitter in sampling intervals and non-negligible and varying latencies.

3.1 Timing parameters

The basic timing parameters of a control task are shown in Fig. 1. It is assumed that the control task is *released* (i.e., inserted into the ready queue of the real-time operating system) periodically at times given by $t_k = hk$, where h is the *nominal sampling interval* of the controller. Due to preemption and blocking from other tasks in the system, the actual *start* of the task may be delayed for some time L_s. This is called the *sampling latency* of the controller. A dynamic

scheduling policy will introduce variations in this interval. The *sampling jitter* is quantified by the difference between the maximum and minimum sampling latencies in all task instances,

$$J_s \stackrel{\text{def}}{=} \max_k L_s^k - \min_k L_s^k. \tag{1}$$

Normally, it can be assumed that the minimum sampling latency of a task is zero, in which case we have $J_s = \max_k L_s^k$. Jitter in the sampling latency will also introduce jitter in the sampling interval h. From the figure, it is seen that the actual sampling interval in period k is given by

$$h^k = h - L_s^{k-1} + L_s^k. \tag{2}$$

The *sampling interval jitter* is quantified by

$$J_h \stackrel{\text{def}}{=} \max_k h^k - \min_k h^k. \tag{3}$$

We can see that the sampling interval jitter is upper bounded by

$$J_h \leq 2J_s. \tag{4}$$

After some computation time and possibly further preemption from other tasks, the controller will actuate the control signal. The delay from the sampling to the actuation is called the *input-output latency*, denoted L_{io}. Varying execution times or task scheduling will introduce variations in this interval. The *input-output jitter* is quantified by

$$J_{io} \stackrel{\text{def}}{=} \max_k L_{io}^k - \min_k L_{io}^k. \tag{5}$$

It is well known that a constant input-output latency decreases the phase margin of the control system, and that it introduces a fundamental limitation on the achievable closed-loop performance. The resulting sampled-data system is time invariant and of finite order, which allows standard linear time-invariant (LTI) analysis to be used (see, e.g., [3]). For a given value of the latency, it is easy to predict the performance degradation due to the delay. Furthermore, it is straightforward to account for a constant latency in most control design methods. From this perspective, a constant input-output latency is preferable over a varying latency.

The scheduling-induced input-output latency of a single control task can be reduced by assigning it a higher priority (or, alternatively, under deadline-based scheduling, a shorter deadline). Of course, this approach will not work for the whole task set.

Another option is to use non-preemptive scheduling. This will guarantee that, once the task has started its execution, it will continue uninterrupted until the end. The disadvantages of this approach are that the scheduling analysis for non-preemptive scheduling is quite complicated (e.g., [19, 27]),

and that the schedulability of other tasks may be compromised. However, as computing speed increases, and, hence, $C \ll T$, it becomes increasingly interesting to execute the tasks with hard deadlines non-preemptively.

A standard way to achieve a short input-output latency in a control task is to separate the algorithm calculations in two parts: *Calculate Output* and *Update State*. Calculate Output contains only the parts of the algorithm that make use of the current sample information. Update State contains the update of the controller states and precalculations for the next period. Update State can therefore be executed after the output signal transmission, hence, reducing the input-output latency. Further improvements can be obtained by scheduling the two parts as subtasks with different priorities, see [6].

A control system with a time-varying input-output latency is quite difficult to analyze, since the standard tools for LTI systems cannot be used. If the statistical properties of the latency are known, then theory from jump linear systems can be used to evaluate the stability and performance of the system (in the mean sense), see [25]. Often, it is not possible to have exact knowledge of the input-output latency distribution. A simple, sufficient stability test for systems where only the range of the latency is known is given in [18]. Assuming zero sampling jitter, the test can guarantee stability for *any* input-output latencies in a given interval (whether they are time-varying, dependent, etc.).

4 Design Approaches

The temporal non-determinism caused by the implementation platform can be approached in two different ways:

- the hard real-time approach, or
- the soft, control-based approach.

The hard real-time approach strives to maximize the temporal determinism by using special-purpose hardware, software, and protocols. This includes techniques such as static cyclic scheduling, time-triggered computing and communication [20], synchronous programming languages [4], and computing models such as Giotto [17]. This approach has several advantages, especially for safety-critical applications. For example, it simplifies attempts at formal verification. The approach also has drawbacks. It has strong requirements on the availability of realistic worst-case bounds on resource utilization, something which is very difficult to obtain in practice. A result of this could be underutilization and, possibly, poor control performance due to sampling intervals that are too long. The approach also makes it difficult to use general-purpose implementation platforms. This is particularly serious, since it is these systems that have the most advantageous price-performance development.

The soft, control-based approach instead views the temporal non-determinism caused by the implementation platform as an uncertainty or disturbance acting on the control loop and handles it using control-based ap-

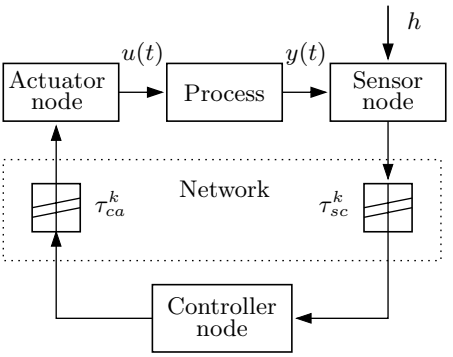


Fig. 2. Distributed digital control system with network communication delays τ_{sc}^k and τ_{ca}^k. From [25].

proaches. This can be done using a number of techniques. The simplest way is to rely on the inherent robustness of feedback. It is well known that feedback increases the robustness towards plant variations. The same holds for variations caused by the implementation platform, i.e., *temporal robustness*. Another approach to deal with jitter in the control design is to explicitly design the controller to be robust, i.e., treat the delay as a parametric uncertainty. Many robust design methods are available, such as H_∞, quantitative feedback theory (QFT), and μ-design. The majority of these methods are developed for plant uncertainties. Although parts of the results carry over to temporal robustness, it is likely that there is room for much more research here.

It is also possible to let the controller actively compensate for the delay in each sample. This can be compared to traditional gain-scheduling and feedforward from disturbances. An optimal, jitter-compensating controller was developed in [25]. The controller compensates for time-varying delays in a control loop, which is closed over a communication network. The setup is shown in Fig. 2. The sensor node samples the process periodically, sending the measurements over the network to the controller node. The controller node is event driven and computes a new control signal as soon as a measurement arrives. The control signal is sent to the event-driven actuator node, which outputs the signal to the process. The linear-quadratic (LQ) state feedback control law has the form

$$u(k) = -L(\tau_{sc}^k) \begin{bmatrix} x(k) \\ u(k-1) \end{bmatrix}, \tag{6}$$

where the feedback gain L depends on the sensor-to-controller delay τ_{sc}^k in the current sample. The computation of the gain vector L is quite involved and requires that the probability distributions of τ_{sc} and τ_{ca} be known. The state feedback can be combined with an optimal state observer that takes the actual delays into account.

The above approach cannot be directly applied to scheduling-induced delays. The problem is that the delay in the current sample will not be known until the task has finished, and by then it is too late to compensate. A simple scheme that compensates for delay in the previous sample is presented in [21]. The compensator has the same basic structure as the well-known Smith predictor, but allows for a time-varying delay.

Many other heuristic jitter compensation schemes have been suggested; see, e.g., [1,13,24]. What the approaches have in common is that they require language or operating system support for instrumenting an application with measurement code.

In order to fully apply these techniques, it is necessary to increase the understanding of how temporal non-determinism affects control performance. This requires new theories and tools that are now beginning to emerge. An important issue that is still lacking is a theory that allows us to determine which level of temporal determinism a given control loop really requires in order to meet given control objectives on stability and performance. Is it necessary to use a time-triggered approach, or will an event-based approach perform satisfactorily? How large are the input-output latencies that can be tolerated? Is it OK to now and then skip a sample in order to maintain the schedulability of the task set? Ideally, one would like to have an index that decides the required level of temporal determinism through a single quantitative measure. One possible name for such an index would be the schedulability margin. This measure would need to combine both a margin with respect to input-output latency and jitter and a margin that decides how large a sampling jitter the loop can tolerate. For constant input-output latencies the classical phase margin can be applied.

An extension of the classical delay margin to time-varying delays is proposed in [10]. The jitter margin $J_m(L)$ is defined as the largest input-output jitter for which closed-loop stability is guaranteed for any time-varying latency $\Delta \in [L, L + J_m(L)]$, where L is the constant part of the input-output latency. The jitter margin is based on the stability theorem defined in [18]. The jitter margin can be used to derive hard deadlines that guarantee closed-loop stability, provided that the scheduling method employed can provide bounds on the worst-case and best-case response times of the controller tasks.

What is still missing in order to be able to define a reasonable analytical concept for a schedulability margin is a simple sampling jitter criterion. The criterion should ideally tell the size of the variations around a nominal sampling interval that the process can tolerate, maintaining stability and acceptable performance.

4.1 Functional robustness

In addition to being temporally robust, it is also important for a control system to be robust towards faults. Numerous theories and methods have been developed for fault detection, diagnosis, and fault tolerance within the

control community. However, the majority of this work concerns faults that occur within the plant, sensors, or actuators. As most software engineers are sadly aware, faults in the software system are far more common than faults in the plant under control. In spite of this, the amount of work that considers robustness against these types of faults, i.e., *functional robustness*, is very small. In [12] a method is presented that renders a control system more robust to computer-level faults leading to data errors. The method is based on the introduction of artificial signal limits in combination with an anti-windup scheme. Related to this, in [2], a methodology is developed for analyzing the impact that these types of data errors have on control system dependability.

5 Codesign Tools

In order for codesign of control and computing systems to become feasible, it is necessary to have software tools that allow the designers to analyze and simulate how timing affects control performance. Such tools have recently begun to emerge, e.g., [11,23,26]. Here, two such tools will be briefly described: Jitterbug (http://www.control.lth.se/~lincoln/jitterbug) and TrueTime (http://www.control.lth.se/~dan/truetime).

5.1 Jitterbug

Jitterbug [8,9,22] is a MATLAB-based toolbox that computes a quadratic performance criterion for a linear control system under various timing conditions. The tool can also compute the spectral density of the signals in the system. Using the toolbox, one can easily and quickly assert how sensitive a control system is to delay, jitter, lost samples, etc., without resorting to simulation. The tool is quite general and can also be used to investigate jitter-compensating controllers, aperiodic controllers, and multi-rate controllers. The main contribution of the toolbox, which is built on well-known theory (linear quadratic Gaussian (LQG) theory and jump linear systems), is to make it easy to apply this type of stochastic analysis to a wide range of problems.

Jitterbug offers a collection of MATLAB routines that allow the user to build and analyze simple timing models of computer-controlled systems. A control system is built by connecting a number of continuous- and discrete-time systems. For each subsystem, optional noise and cost specifications may be given. In the simplest case, the discrete-time systems are assumed to be updated in order during the control period. For each discrete system, a random delay (described by a discrete probability density function) can be specified that must elapse before the next system is updated. The total cost of the system (summed over all subsystems) is computed algebraically if the timing model system is periodic or iteratively if the timing model is aperiodic.

To make the performance analysis feasible, Jitterbug can only handle a certain class of system. The control system is built from linear systems driven

by white noise, and the performance criterion to be evaluated is specified as a quadratic, stationary cost function. The timing delays in one period are assumed to be independent from the delays in the previous period. Also, the delay probability density functions are discretized using a time-grain that is common to the whole model.

Even though a quadratic cost function can hardly capture all aspects of a control loop, it can still be useful when one wants to quickly judge several possible controller implementations against each other. A higher value of the cost function typically indicates that the closed-loop system is less stable (i.e., more oscillatory), and an infinite cost means that the control loop is unstable. The cost function can easily be evaluated for a large set of design parameters and can be used as a basis for the control and real-time design.

Jitterbug models

In Jitterbug, a control system is described by two parallel models: a signal model and a timing model. The signal model is given by a number of connected, linear, continuous- and discrete-time systems. The timing model consists of a number of timing nodes and describes when the different discrete-time systems should be updated during the control period.

An example is shown in Fig. 3, where a computer-controlled system is modeled by four blocks. The plant is described by the continuous-time system G, and the controller is described by the three discrete-time systems H_1, H_2, and H_3. The system H_1 could represent a periodic sampler, H_2 could represent the computation of the control signal, and H_3 could represent the actuator. The associated timing model says that, at the beginning of each period, H_1 should first be executed (updated). Then there is a random delay τ_1 until H_2 is executed, and another random delay τ_2 until H_3 is executed. The delays could model computational delays, scheduling delays, or network transmission delays.

The same discrete-time system may be updated in several timing nodes. It is possible to specify different update equations in the various cases. This can be used to model a filter where the update equations look different depending on whether or not a measurement value is available. It is also possible to make the update equations depend on the time since the first node became active. This can be used to model jitter-compensating controllers, for example.

For some systems, it is desirable to specify alternative execution paths (and thereby multiple next nodes). In Jitterbug, two such cases can be modeled (see Fig. 4):

(a) A vector n of next nodes can be specified with a probability vector p. After the delay, execution node $n(i)$ will be activated with probability $p(i)$. This can be used to model a sample being lost with some probability.
(b) A vector n of next nodes can be specified with a time vector t. If the total delay in the system since the node exceeds $t(i)$, node $n(i)$ will be

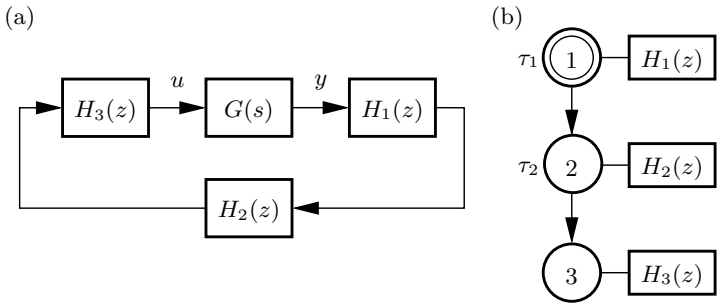


Fig. 3. A simple Jitterbug model of a computer-controlled system: (a) signal model and (b) timing model. The process is described by the continuous-time system $G(s)$ and the controller is described by the three discrete-time systems $H_1(z)$, $H_2(z)$, and $H_3(z)$, representing the sampler, the control algorithm, and the actuator. The discrete systems are executed according to the periodic timing model.

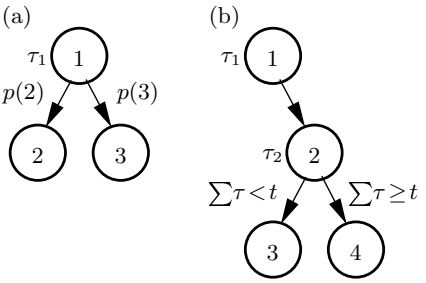


Fig. 4. Alternative execution paths in a Jitterbug execution model: (a) random choice of path and (b) choice of path depending on the total delay from the first node.

activated next. This can be used to model time-outs and various compensation schemes.

Example. The Jitterbug commands used to define the control system of Fig. 3 are given in Fig. 5. The process is modeled by the continuous-time system

$$G(s) = \frac{1000}{s(s+1)},$$

and the controller is a discrete-time PD controller implemented as

$$H_2(z) = -K\left(1 + \frac{T_d}{h}\frac{z-1}{z}\right).$$

The sampler and the actuator are described by the trivial discrete-time systems

$$H_1(z) = H_3(z) = 1.$$

```
G = 1000/(s*(s+1));                          Define the process
H1 = 1;                                      Define the sampler
H2 = -K*(1+Td/h*(z-1)/z);                    Define the controller
H3 = 1;                                      Define the actuator
Ptau1 = [ ... ];              Define delay probability distribution 1
Ptau2 = [ ... ];              Define delay probability distribution 2
N = initjitterbug(delta,h);                  Set time-grain and period
N = addtimingnode(N,1,Ptau1,2);              Define timing node 1
N = addtimingnode(N,2,Ptau2,3);              Define timing node 2
N = addtimingnode(N,3);                      Define timing node 3
N = addcontsys(N,1,G,4,Q,R1,R2);     Add plant, specify cost and noise
N = adddiscsys(N,2,H1,1,1);                  Add sampler to node 1
N = adddiscsys(N,3,H2,2,2);                  Add controller to node 2
N = adddiscsys(N,4,H3,3,3);                  Add actuator to node 3
N = calcdynamics(N);                         Calculate internal dynamics
J = calccost(N);                             Calculate the total cost
```

Fig. 5. This MATLAB script shows the commands needed to compute the performance index of the control system defined by the timing and signal models in Fig. 3.

The delays in the computer system are modeled by the two (possibly random) variables τ_1 and τ_2. The total delay from sampling to actuation is thus given by $\tau_1 + \tau_2$.

Using the defined Jitterbug model, it is straightforward to investigate, e.g., how sensitive the control loop is to slow sampling and constant delays (by sweeping over suitable ranges for these parameters) and random delays with jitter compensation. For more details and other illustrative examples (including multi-rate control, overrun handling, and notch filter implementations), see [9].

5.2 TrueTime

TrueTime [8, 15, 16] is a MATLAB/Simulink-based tool that facilitates simulation of the temporal behavior of a multi-tasking real-time kernel executing controller tasks. The tasks are controlling processes that are modeled as ordinary continuous-time Simulink blocks. TrueTime also makes it possible to simulate simple models of communication networks and their influence on networked control loops.

In TrueTime, kernel and network Simulink blocks are introduced, the interfaces of which are shown in Fig. 6. The kernel blocks are event driven and execute code that models, e.g., I/O tasks, control algorithms, and network interfaces. The scheduling policy of the individual kernel blocks is arbitrary and decided by the user. Likewise, in the network, messages are sent and received according to a chosen network model.

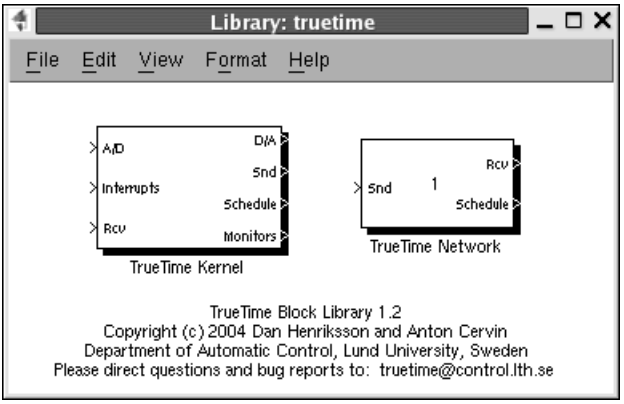


Fig. 6. The TrueTime block library. The Schedule and Monitor outputs display the allocation of common resources (CPU, monitors, network) during the simulation.

The level of simulation detail is also chosen by the user—it is often neither necessary nor desirable to simulate code execution on instruction level or network transmissions on bit level. TrueTime allows the execution time of tasks and the transmission times of messages to be modeled as constant, random, or data dependent. Furthermore, TrueTime allows simulation of context switching and task synchronization using events or monitors.

TrueTime can be used as an experimental platform for research on dynamic real-time control systems. For instance, it is possible to study compensation schemes that adjust the control algorithm based on measurements of actual timing variations (i.e., to treat the temporal uncertainty as a disturbance and manage it with feedforward or gain scheduling). It is also easy to experiment with more flexible approaches to real-time scheduling of controllers, such as feedback scheduling, see [7]. There the available CPU or network resources are dynamically distributed according to the current situation (CPU load, the performance of the different loops, etc.) in the system.

The kernel block

The kernel block is a MATLAB S-function that simulates a computer with a simple but flexible real-time kernel, analog-to-digital (A/D) and digital-to-analog (D/A) converters, a network interface, and external interrupt channels. The kernel executes user-defined tasks and interrupt handlers. Internally, the kernel maintains several data structures that are commonly found in a real-time kernel: a ready queue, a time queue, and records for tasks, interrupt handlers, monitors, and timers that have been created for the simulation.

An arbitrary number of tasks can be created to run in the TrueTime kernel. Tasks may also be created dynamically as the simulation progresses. Tasks are used to simulate both periodic activities, such as controller and I/O tasks, and

aperiodic activities, such as communication tasks and event-driven controllers. Aperiodic tasks are executed by the creation of task instances (jobs).

Each task is characterized by a number of static (e.g., relative deadline, period, and priority) and dynamic (e.g., absolute deadline and release time) attributes. In accordance with the Real-Time Specification for Java (RTSJ) [5], it is furthermore possible to attach two overrun handlers to each task: a deadline overrun handler (triggered if the task misses its deadline) and an execution time overrun handler (triggered if the task executes longer than its worst-case execution time).

Interrupts may be generated in two ways: externally (associated with the external interrupt channel of the kernel block) or internally (triggered by user-defined timers). When an external or internal interrupt occurs, a user-defined interrupt handler is scheduled to serve the interrupt.

The execution of tasks and interrupt handlers is defined by user-written code functions. These functions can be written either in C++ (for speed) or as MATLAB m-files (for ease of use). Control algorithms may also be defined graphically using ordinary discrete Simulink block diagrams.

Simulated execution occurs at three distinct priority levels: the interrupt level (highest priority), the kernel level, and the task level (lowest priority). The execution may be preemptive or non-preemptive; this can be specified individually for each task and interrupt handler.

At the interrupt level, interrupt handlers are scheduled according to fixed priorities. At the task level, dynamic-priority scheduling may be used. At each scheduling point, the priority of a task is given by a user-defined priority function, which is a function of the task attributes. This makes it easy to simulate different scheduling policies. For instance, a priority function that returns a priority number implies fixed-priority scheduling, whereas a priority function that returns the absolute deadline implies earliest-deadline-first scheduling. Predefined priority functions exist for rate-monotonic, deadline-monotonic, fixed-priority, and earliest-deadline-first scheduling.

The network block

The network block is event driven and executes when messages enter or leave the network. When a node tries to transmit a message, a triggering signal is sent to the network block on the corresponding input channel. When the simulated transmission of the message is finished, the network block sends a new triggering signal on the output channel corresponding to the receiving node. The transmitted message is put in a buffer at the receiving computer node.

A message contains information about the sending and the receiving computer node, arbitrary user data (typically measurement signals or control signals), the length of the message, and optional real-time attributes such as a priority or a deadline.

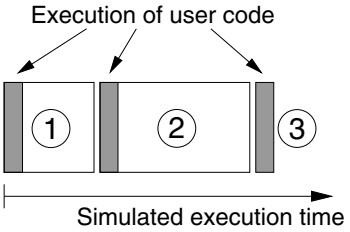


Fig. 7. The execution of the code associated with tasks and interrupt handlers is modeled by a number of code segments with different execution times. Execution of user code occurs at the beginning of each code segment.

The network block simulates medium access and packet transmission in a local area network. Six simple models of networks are currently supported: carrier-sense multiple access/collision detection (CSMA/CD) (e.g., Ethernet), carrier-sense multiple access/arbitration on message priority (CSMA/AMP) (e.g., CAN), Round Robin (e.g., Token Bus), frequency-division multiplexed access (FDMA), time-division multiplexed access (TDMA) (e.g., time-triggered protocol (TTP)), and switched Ethernet. The propagation delay is ignored, since it is typically very small in a local area network. Only packet-level simulation is supported, i.e., it is assumed that higher protocol levels in the kernel nodes have divided long messages into packets.

Configuring the network block involves specifying a number of general parameters, such as transmission rate, network model, and probability for packet loss. Protocol-specific parameters that need to be supplied include the time slot and cyclic schedule in the case of TDMA.

Execution model

The execution of tasks and interrupt handlers is defined by code functions. A code function is further divided into code segments according to the execution model in Fig. 7. The code can interact with other tasks and with the environment at the beginning of each code segment. This execution model makes it possible to model input-output latencies, blocking when accessing shared resources, etc. The number of segments can be chosen to simulate an arbitrary time granularity of the code execution. Technically, it would be possible to simulate very fine-grained details occurring at the machine instruction level, such as race conditions. However, that would require a large number of code segments.

The simulated execution time of each segment is returned by the code function, and can be modeled as constant, random, or even data dependent. The kernel keeps track of the current segment and calls the code functions with the proper argument during the simulation. Execution resumes in the

```
function [exectime, data] = Pcontroller(segment, data)
switch segment,
  case 1,
    r = ttAnalogIn(1);
    y = ttAnalogIn(2);
    data.u = data.K*(r-y);
    exectime = 0.001;
  case 2,
    ttAnalogOut(1, data.u);
    exectime = 0.001;
  case 3,
    exectime = -1; % finished
end
```

Fig. 8. Example of a standard code function written in MATLAB code. The local memory of the controller task is represented by the data structure `data`. This stores the controller gain and the control signal between invocations of different code segments.

next segment when the task has been running for the time associated with the previous segment. This means that preemption by higher-priority activities and interrupts may cause the actual delay between execution of segments to be longer than the execution time.

Fig. 8 shows an example of a code function corresponding to the time line in Fig. 7. The function implements a standard P controller. In the first segment, the plant is sampled and the control signal is computed. In the second segment, the control signal is actuated and the controller states are updated. The third segment indicates the end of execution by returning a negative execution time.

The data structure `data` represents the local memory of the task and is used to store the control signal and measured variable between calls to the different segments. A/D and D/A conversion is performed using the kernel primitives `ttAnalogIn` and `ttAnalogOut`.

Note that the input-output latency of this controller will be *at least* 2 ms (i.e., the execution time of the first segment). However, if there is preemption from other high-priority tasks, the actual input-output latency will be longer.

6 Conclusion

This chapter has discussed the relationships between control design and the real-time scheduling, and how implementation-level timing variations can be handled in the control design. For embedded applications with limited computing resources, this type of implementation-aware control is especially important. Designing a real-time control system is essentially a codesign problem. Choices made in the real-time design will affect the control design and

vice versa. For instance, deciding on a particular network protocol will give rise to certain delay distributions that must be taken into account in the controller design. On the other hand, bandwidth requirements in the control loops will influence the choice of CPU and network speed. Using an analysis tool such as Jitterbug, one can quickly assert how sensitive the control loop is to slow sampling rates, delay, jitter, and other timing problems. Aided by this information, the user can proceed with more detailed, system-wide real-time and control design using a simulation tool such as TrueTime.

References

1. Pedro Albertos and Alfons Crespo. Real-time control of non-uniformly sampled systems. *Control Engineering Practice*, 7:445–458, 1999.
2. Örjan Askerdal, Magnus Gäfvert, Martin Hiller, and Neeraj Suri. Analyzing the impact of data errors in safety-critical control systems. *IEICE Transactions on Information and Systems*, E86-D(12), December 2003. Special Issue on Dependable Computing.
3. Karl Johan Åström and Björn Wittenmark. *Computer-Controlled Systems*. Prentice Hall, Upper Saddle River, NJ, third edition, 1997.
4. A. Benveniste and G. Berry. The synchronous approach to real-time programming. *Proceedings of the IEEE*, 79:1270–1282, 1991.
5. G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, and M. Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, Reading, MA, 2000.
6. Anton Cervin. Improved scheduling of control tasks. In *Proceedings of the 11th Euromicro Conference on Real-Time Systems*, pages 4–10, York, U.K., June 1999.
7. Anton Cervin, Johan Eker, Bo Bernhardsson, and Karl-Erik Årzén. Feedback-feedforward scheduling of control tasks. *Real-Time Systems*, 23(1–2):25–53, July 2002.
8. Anton Cervin, Dan Henriksson, Bo Lincoln, Johan Eker, and Karl-Erik Årzén. How does control timing affect performance? *IEEE Control Systems Magazine*, 23(3):16–30, June 2003.
9. Anton Cervin and Bo Lincoln. Jitterbug 1.1—Reference manual. Technical Report ISRN LUTFD2/TFRT--7604--SE, Department of Automatic Control, Lund Institute of Technology, Sweden, January 2003.
10. Anton Cervin, Bo Lincoln, Johan Eker, Karl-Erik Årzén, and Giorgio Buttazzo. The jitter margin and its application in the design of real-time control systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications*, Göteborg, Sweden, August 2004.
11. Jad El-khoury and M. Törngren. Towards a toolset for architectural design of distributed real-time control systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, London, England, December 2001.
12. Magnus Gäfvert, Björn Wittenmark, and Örjan Askerdal. On the effect of transient data-errors in controller implementations. In *Proceedings of the American Control Conference*, pages 3411–3416, Denver, CO, 2003.

13. Tore Hägglund. A predictive PI controller for processes with long dead times. *IEEE Control Systems Magazine*, 12(1):57–60, 1992.
14. H. Hanselmann. Implementation of digital controllers—A survey. *Automatica*, 23(1):7–32, 1987.
15. Dan Henriksson and Anton Cervin. TrueTime 1.1—Reference manual. Technical Report ISRN LUTFD2/TFRT--7605--SE, Department of Automatic Control, Lund Institute of Technology, October 2003.
16. Dan Henriksson, Anton Cervin, and Karl-Erik Årzén. TrueTime: Simulation of control loops under shared computer resources. In *Proceedings of the 15th IFAC World Congress on Automatic Control*, Barcelona, Spain, July 2002.
17. T. Henzinger, B. Horowitz, and C. Kirsch. Giotto: a time-triggered language for embedded programming. *Proceedings of the IEEE*, 91(1):84–99, 2003.
18. Chung-Yao Kao and Bo Lincoln. Simple stability criteria for systems with time-varying delays. *Automatica*, 40(8):1429–1434, 2004.
19. M. H. Klein, T. Ralya, B. Pollak, R. Obenza, and M. Gonzalez Hrbour. *A Practitioner's Handbook for Real-Time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems*. Kluwer Academic Publishers, Dordrecht, 1993.
20. H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.
21. Bo Lincoln. Jitter compensation in digital control systems. In *Proceedings of the 2002 American Control Conference*, May 2002.
22. Bo Lincoln and Anton Cervin. Jitterbug: A tool for analysis of real-time control performance. In *Proceedings of the 41st IEEE Conference on Decision and Control*, Las Vegas, NV, December 2002.
23. Jie Liu and Edward Lee. Timed multitasking for real-time embedded software. *IEEE Control Systems Magazine*, 23(1), February 2003.
24. Pau Marti, Gerhard Fohler, Krithi Ramamritham, and Josep M. Fuertes. Jitter compensation for real-time control systems. In *Proceedings of the 22nd IEEE Real-Time Systems Symposium*, 2001.
25. Johan Nilsson, Bo Bernhardsson, and Björn Wittenmark. Stochastic analysis and control of real-time systems with random time delays. *Automatica*, 34(1):57–64, 1998.
26. L. Palopoli, G. Lipari, G. Lamastra, and L. Abeni. An object-oriented tool for simulating distributed real-time control systems. *Software—Practice and Experience*, 32:907–932, 2002.
27. John A. Stankovic, Marco Spuri, Krithi Ramamritham, and Giorgio C. Buttazzo. *Deadline Scheduling for Real-Time Systems—EDF and Related Algorithms*. Kluwer Academic Publishers, Dordrecht, 1998.

From Control Loops to Real-Time Programs

Paul Caspi and Oded Maler

Verimag-CNRS
2, av. de Vignate
38610 Gires
France
www-verimag.imag.fr
Paul.Caspi@imag.fr
Oded.Maler@imag.fr

1 Introduction

This article discusses what we consider one of the central aspects of embedded systems: the realization of control systems by software. Although computers are today the most popular medium for implementing controllers, we feel that the state of understanding of this topic is not satisfactory, mostly due to the fact that it is situated in the frontier between two different cultures and world views (*control* and *informatics*) which are not easy to reconcile. The purpose of this article is to clarify these issues and present them in a uniform and, hopefully, coherent manner.

The article is organized as follows. We start with a short high-level discussion of the two phenomena involved, *control* and *computation*. In Section 2 we explain the basic issues related to the realization of controllers by software using a simple proportional-integral-derivative (PID) controller as an example. In Section 3 we move to more complex multi-periodic control loops and describe various approaches for scheduling them on a sequential computer. Section 4 is devoted to discrete-event (and hybrid) systems and their software implementation. Finally, in Section 5 we briefly discuss distributed control and fault tolerance.

1.1 Control

A controller is a mechanism that interacts with part of the world (the "plant") by measuring certain variables and exerting some influence in order to steer it toward desirable states. The rule that determines what the controller does as a function of what it observes (and of its own state) is called the feedback function. In early days of control, the feedback function was "computed" mechanically: for example, in the famous Watt governor, analyzed mathemati-

cally by Maxwell, the angle of the governor was determined by the angular velocity by purely mechanical means.

With the advent of electronics, the process of computing that function was decoupled from measurement and actuation. Physical magnitudes of different natures were transformed into low-power electric signals. These signals were fed into an analog computer whose output signals were converted into physical quantities and fed back to the plant. From a mathematical standpoint, this architecture posed no conceptual problems. The underlying model of the plant and of the analog computer were of the same nature. The former was a continuous dynamical system with evolution defined by the differential equations of the corresponding physical theory (mechanics, thermodynamics, etc.), and the latter consisted of an electrical circuit with dynamics governed by similar types of laws. Schematically, we can define the evolution of the plant by the equation $\dot{x} = f(x, d, u)$ with x being the state of the plant, d some external disturbance and u the control signal. The dynamics of the controller implemented by an analog circuit can be likewise written as $\dot{u} = g(u, x, x_0)$, with x_0 being a reference signal, and the evolution of the controlled plant is obtained by the composition of these two equations. This is the conceptual framework underlying classical control theory, where the feedback function is "computed" continuously at each and every time instant.

The introduction of digital computers changed this picture completely. To start with, the computation of a function by digital means is an inherently discrete process. Numbers are represented by binary encoding rather than by physical magnitudes. Consequently, sensor readings should be transformed from analog to digital representation before the computation; conversely, the results of the computation should be transformed back from digital to analog form. The computation is done by a sequence of discrete steps that take time, and the electrical values on different wires are meaningless until the computation terminates. Thus it makes no sense to connect the computer to the plant in a continuous manner. The transition from physical to digital control is illustrated in Fig. 1.

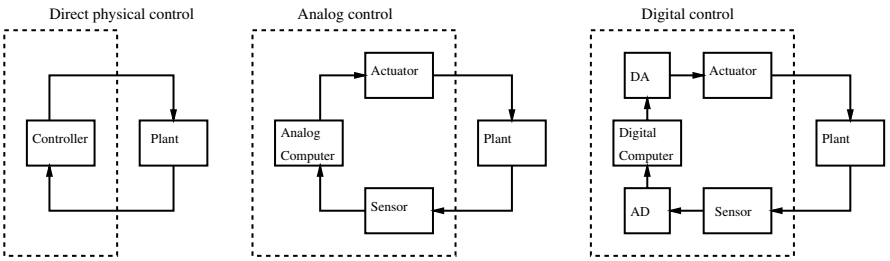


Fig. 1. From physical to analog to digital control

To cope with these changes a comprehensive theory of digital "sampled" control has been developed [2]. Within this theory, the interaction of the controller and the plant is restricted to sampling points, a (typically periodic) discrete subset of the real-time axis. At these points sensors are read, and the values are digitized and handed over to the computer, which computes the value of the feedback function, which is converted to analog and fed back to the plant via the actuators. From the control point of view, the sampling rate is determined by the dynamics of the plant, with the obvious intuition that a faster and more complex dynamics requires more frequent sampling. The sampling period is determined by the desired level of approximation and by the properties of the signal.

The role of the computer in this collaboration is to be able to compute the value of the function (including the analog-to-digital (A/D) and digital-to-analog (D/A) conversions) fast enough, that is, between two sampling points. Once this is guaranteed, the control engineer can regard the computer as yet another (discrete-time) block in the system and ignore its "computerhood." This is certainly true for simple single-input-single-output (SISO) systems, but becomes less and less so when the structure of the control loops becomes more complex. Before discussing these issues, let us take a look at computation.

1.2 Computation

In the early days of digital computers, their interaction with the outside world was rather limited. A typical batch program for producing a payroll or for performing an intensive numerical computation did not interact with the external world during execution. Such systems, termed "transformational" systems by Harel and Pnueli [12], read their input at the beginning, embark on the computation process and output the result upon termination. The fundamental theories of computability and complexity are tailored to this type of "autistic" computation. They can say which types of function from input to output can be computed at all, and for those that can, how the number of computation steps grows asymptotically with the size of the problem.

If we insist on philosophical rigor, we must admit that even computations of this type are "embedded" in some sort of a larger process. The batch numerical computation could have been, for example, a finite element algorithm to determine the stability of a building. Such a computation is part of the construction process and should be invoked each time a new building is designed or when a change is initiated by the architect. The computation time of such a program, even in the early days when it was measured by hours and days, was still reasonable with respect to the time scale of a typical construction project. Likewise, a payroll program is part of the "control loop" of an organization which reads the time sheet of the employees and prints checks at the end of the month. If the execution time of such a program were on the order of magnitude of weeks, it could not fulfill its role in that control loop. So

the difference with respect to the progressively more interactive computations that will be described in the sequel is also a quantitative matter of time scales.

With the development of time-sharing operating systems, the nature of computation became more interactive. A typical example would be a text editor, a command shell or any other program interacting with one or more users via keyboards and screens.[1] What is the function that such an interactive program "computes"? People familiar with automata theory can see that it is a *sequential function*, something that transforms sequences of input symbols (commands) to sequences of output symbols (responses). The important point in such functions is that the process of computation is no longer isolated from the input/output process but is rather interleaved with it: the user types a command, the computer computes a response (and possibly changes its internal state) and so on. These are called "reactive" systems in [12].

While such interactive systems differ considerably from batch programs that operate within a static environment which does not change during computation, they still operate under certain restricting assumptions concerning their environment, which is typically a human user or a computer program that follows some protocol. The implicit assumption is that the environment behaves in a manner similar to a player in a turn-based game like chess; that is, the user waits for the response of the computer before entering the next input. As in the case of batch systems, this metaphor is valid as long as the computer is not slower than the external environment against which it works. When a person's typing speed exceeds the reaction speed of the text editor, or when a transmitter transmits faster than a receiver receives, everything breaks down.

Digital implementations of continuous control systems, the subject of this chapter, interact with the physical world, a player which is assumed to be governed by differential equations, and which evolves independently of whether the computer is ready to interact with it. Of course, in the same way as a text editor may ignore characters that are typed too fast, a slow computer may ignore sensor readings or not update actuator values fast enough. However, in many "time-critical" systems, the ability of the computer to meet the rhythm of the environment is *the key to the usefulness of the system.* Failing to do so may lead in some cases to catastrophic results, and in others, to severe degradation in performance. Such systems are often called real-time systems to distinguish them from the types of programs previously described and to indicate the tight coupling between the internal time inside the computer and the time of the external world.[2] In the next section we discuss various differences between such programs and the control loops that they realize.

[1] Today it is hard to imagine how computing could be otherwise, but the passage from batch- to terminal-based computation was revolutionary at the time, and the authors are old enough to remember that.

[2] Sometimes the terms online versus offline are used for similar purposes.

2 From Mathematical Descriptions to Programs

Programs implementing control systems differ from their corresponding discrete-time recurrence equations in several aspects, the first of which is not particular to control systems but is concerned with different levels of abstraction in which algorithms can be described. For instance, algorithms for searching directed graphs may be defined in terms of the abstract structure of the graph, the mathematical $G = (V, E)$, without paying attention to the way the graph is stored in memory. An abstract algorithm may contain statements such as "for each successor of a vertex v do" without being explicit about the way the successors of a node are retrieved from the data structure representing the graph. More concrete programs, written in languages such as C, need to specify these details. Between these levels and the actual physical realization there are many intermediate levels (assembly and machine code, microcode, architecture, etc.) and one of the great achievements of computer science and engineering is that most of the transformations between these levels are done automatically using computer programs.

As an illustrative example we consider one of the most popular forms of control, the PID controller, and see how it is transformed into a program. An important feature of feedback functions is that they are typically dynamical systems by themselves, admitting a *state* which influences their output and future behavior. Fig. 2 shows the Simulink diagram of a typical sampled-data PID controller. The annotation of the Simulink blocks is written in the z-transform formalism, which is a discrete version of a frequency-domain representation of systems, where delay and memory are expressed using the $1/z$ operator. An explanation of this formalism can be found elsewhere in the handbook, and we focus here on a more "mechanical" state-space description of the controller. What a PID controller essentially does is to take the input signal I, compute its derivative D and integral S and then compute the output O as some linear combination of I, S and D. The state variables of the system include the integral S and the previous value of the input J, which is needed for computing the derivative. The following system of recurrence equations defines the semantics of the controller as a set O_n of output sequences whose relation with the input sequence I_n is defined by

$$\begin{aligned} S_{-1} &= I_{-1} = 0.0 \\ S_n &= S_{n-1} + 0.1 \cdot I_n \\ O_n &= 5.8 \cdot I_n + 4 \cdot S_n + 3.8 \cdot 10.0 \cdot (I_n - I_{n-1}). \end{aligned} \tag{1}$$

The first line defines the initial values of state variable S and the second line defines its subsequent value for every $n \geq 0$. The last line determines the output, using $I_n - I_{n-1}$ as the derivative. Since old values of the input are not typically kept in memory, we will need to store this information in an auxiliary state variable J satisfying $J_n = I_n$, and replacing I_{n-1} in the definition of O_n by J_{n-1}.

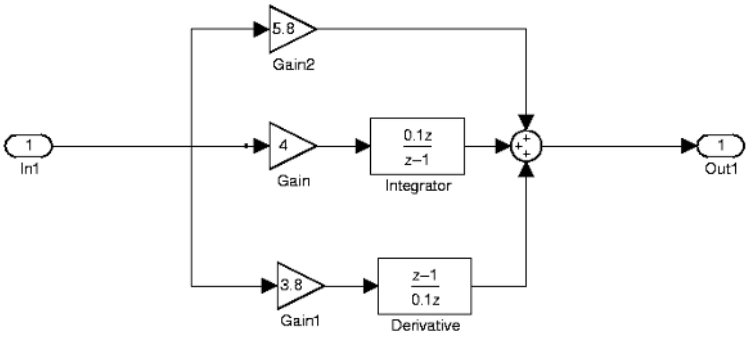


Fig. 2. A PID controller represented by a Simulink block diagram

Before showing the corresponding program, let us note that since (1) involves memory that has to be maintained and propagated between *successive invocations* of the program, the corresponding programming construct is better viewed as a *class* in an object-oriented language such as C^{++} or Java. However, since this point of view is probably not so familiar to most readers, we will realize it as a C program with *global variables*. These variables continue to exist between successive invocations of the program (like latches in sequential digital circuits when the clock signal is low). The program shown in Table 1 is a result of a rather straightforward transformation of (1).

```
/* memories */
float S = 0.0, J = 0.0;

void dispid_cycle (){
  float I,O;
  float J_1,S_1;

  I = Input();

  J_1 = I;
  S_1 = S + 0.1 * I * 4.0;
  O = I * 5.8 + S_1 + 10.0 * 3.8 * (I-J);
  J = J_1;
  S = S_1;

  Output(O);
}
```

Table 1. A program realizing a PID controller

The first part of the program in Table 1 is the declaration and initialization of the global variables J and S. The second part, the dispid_cycle procedure, describes the computation to be performed at each invocation of the program. It uses auxiliary variables J_1 and S_1 into which the new state is computed. The procedure presupposes two auxiliary functions Input and Output provided by the execution platform, which take care of bringing (digitized) sensor inputs into I and writing O onto the actuators. The implementation details of these functions are outside the scope of this article. The computational part of the procedure consists of taking the input and propagating it through a network of computations to produce the output. We first compute the next values of the state variables, then compute the output, write the new state values into the global variables and finally write the output and exit.

Upon closer inspection one can see that we do not really need the auxiliary variable S_1 because only the *new* value of S is used while computing O. Consequently, we can replace the computation of S_1 by direct computation of S, use S in the computation of O and discard the assignment statement S = S_1. In fact, we can do similar things with J, by putting the statement J=I after the computation of the output, to obtain the optimized program in Table 2.

```
/* memories */
float S = 0.0, J = 0.0;

void dispid_cycle (){
  float I,O;

  I = Input();

  S = S +0.1 * I * 4.0;
  O = I * 5.8 + S + 10.0 * 3.8 * (I-J);
  J = I;

  Output(O);
}
```

Table 2. An optimized program for the PID controller

Saving two variables and two assignment statements is not much, but for complex control systems that should run on cheap micro controllers, the accumulated effect of such savings can be significant.

The reader can easily appreciate that the process of writing, modifying and optimizing such programs manually is error prone and that it would be much safer to derive it automatically from the high-level Simulink model. We have derived a program similar to the program in Table 2 from the Simulink model of Fig. 2 in two steps. First, the Simulink-to-Lustre translator [6] was used to transform the model into a program in Lustre, a language [11] which provides

rigorous syntax and semantics for expressing data-flow equations such as (1). Then the Reluc Lustre-to-C code generator [9] produced the program after automatic analysis of state variables, dependencies and other optimizations.

The story does not end with the generation of machine code by the C compiler, as there are some additional conditions associated with the execution platform that need to be met. To begin with, the platform should support the I/O functions and be properly connected to all the machinery for conversion between digital and analog data. Second, the proper functioning of the program depends crucially on its being invoked every T time units, where T is the sampling period of the discrete-time system according to which the parameters of the PID controller were derived. Not adhering to this sampling period may result in a strong deviation of the program behavior from the intended one. This is a very particular (and rather unexpected) class of software errors inherent in control applications.

To ensure the correct periodic activation of the program we need access to a *real-time clock* that will trigger the execution every T time units. But this is not enough due to yet another important difference between an abstract mathematical function and a program that computes it: the former is *timeless* while the latter takes some time to compute. For a program such as `dispid_cycle` to function, the condition $C < T$ should hold, where C is its worst-case execution time (WCET). If this requirement is not met, the program will not terminate before its next invocation (see the timing diagram in Fig. 3). Measuring and estimating the WCET of a program on a given architecture is not an easy task, especially for modern processors, and it is subject to extensive ongoing research [25].

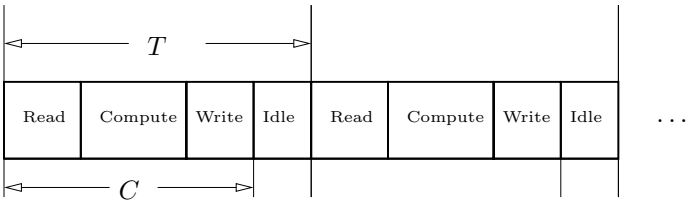


Fig. 3. The execution of a control program with a period T

Once these conditions are fulfilled, several implementation techniques can be used. Historically, such controllers were first implemented on a bare machine, without using any operating system (OS). The real-time clock acts as an interrupt that transfers control to the program. If the scheduling condition $C < T$ is satisfied, this interrupt occurs after the program has terminated and the computer is idle. Hence, unlike preemptive scheduling, there is no need for context switching and complex OS services. This implementation technique is thus both simple and safe and does not need to rely on a complex piece

of software like an OS, which is difficult to validate. Much progress has been made in real-time OS (RTOS) technology, and today commercial systems are available that have been exercised and debugged by a large number of users and can be considered quite safe. Hence the role of monitoring the real-time clock and dispatching the program for execution can be delegated to an OS.

This concludes the discussion on the implementation of simple control programs, where we have tried to touch upon the key relevant computational aspects. In the next section we focus on the timing-related aspects of implementing more complex control loops.

3 Complex Periodic Controllers

In many control applications, systems have several degrees of freedom that must be controlled *simultaneously*. Mathematically each controller c_i is just another recurrence equation that coexists with the other equations. Computationally, these loops should be realized on a *sequential* computer that can do one thing at a time. The problem of how to "sequentialize" and schedule these parallel processes is one of the major topics in real-time systems. It is important that each invocation of a controller has its relevant inputs ready before it starts executing and that the computation of all its outputs and their transmission to the outside world terminate in due time. This is the basic functional requirement from real-time control software, a fact sometimes obscured by details of operating systems and scheduling policies.

3.1 Single period

We start with the simple case where all controllers share the same sampling period T. This means that all of them should be invoked at every cycle of the system. A necessary condition for realizing these controllers sequentially on a given architecture is that all the computations (including input and output) should fit inside the cycle or, in other words, the condition $\sum C_i < T$ is satisfied where each C_i is the WCET of controller c_i on that architecture.

In this setting, the code of each controller can be generated separately as described in Section 2. The sequential implementation of the whole control program can be achieved by a simple scheduler that invokes the controllers one after the other. However, a somewhat less modular but more efficient method consists of gathering all the controllers into a single program and using an optimizing compiler to generate the code of the global controller. By analyzing the structure of the controllers and their data dependencies, a smart code generator can find out that some parts of the computation are shared by several controllers and need not be computed more than once. Such optimizations may reduce the number of operations and a slower computer can be used to achieve the required sampling rate. With the progress of these code generators, this technique is becoming more popular. Verifying the correctness of such optimizing compilers is, by itself, an active research topic.

3.2 Multiple periods

When a system has to control several variables, it is often the case that the variables follow dynamics of different speeds and need to be controlled at different sampling rates. The specification of such a multi-rate system can be given by a collection of pairs $\{(c_i, T_i)\}_{i=1\ldots n}$ where T_i is the period of controller c_i, which can be considered as an integer. With such a task specification we associate two numbers, the basic period $T = gcd(T_1, \ldots, T_n)$, and the super-period $P = lcm(T_1, \ldots, T_n)$, where *gcd* and *lcm* are, respectively, the greatest common divisor and the least common multiple of the task periods. As a running example we consider the 3 task system $S_{123} = \{(c_1, 1), (c_2, 2), (c_3, 3)\}$ with $T = 1$ and $P = 6$, depicted graphically in Fig. 4. The implementation of this abstract specification consists of allocating portions of the timeline of the processor to instances of the controllers (tasks) so that their execution times satisfy the implied constraints. Due to periodicity, if a schedule is found for the first P cycles, it can be repeated indefinitely for the rest of the timeline.

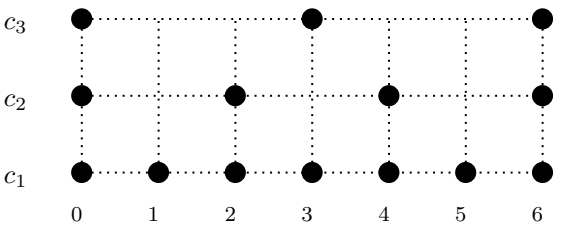


Fig. 4. A multi-rate specification $S_{123} = \{(c_1, 1), (c_2, 2), (c_3, 3)\}$

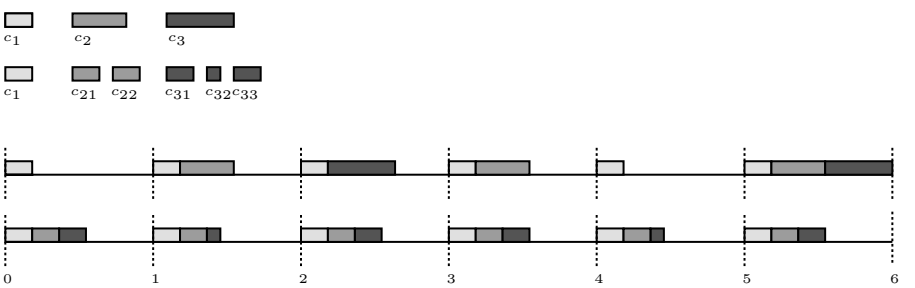


Fig. 5. Schedules for example S_{123}: a simplistic imbalanced schedule versus static partitioning

Cyclic executive

The most straightforward solution is to execute c_1 every cycle, c_2 every second cycle and c_3 every third cycle (see Fig. 5). While this solution is simple and natural, it is not very efficient in utilizing the computer time. As we can see, there are very "busy" cycles where all three controllers need to be executed, while in others the computer is mostly idle. Using this approach, it is the most busy cycle which determines the relation between platform speed and feasibility of the schedule. In this example the schedule is feasible only on platforms satisfying $C_1 + C_2 + C_3 < T$.[3]

More efficient solution schemes are based on the assumption that the nth instance of task c_i can be executed anywhere in the interval $[(n-1) \cdot T_i, n \cdot T_i]$. The lower and upper bounds of the interval are often called, respectively, the *release time* and *deadline* of the task. The set of all such intervals for our example is depicted below:

c_1: $[0,1], [1,2], [2,3], [3,4], [4,5], [5,6]$
c_2: $[0,2], [2,4], [4,6]$
c_3: $[0,3], [3,6]$

Instead of restricting the execution of the slow controllers c_2 and c_3 to the cycle where they need to produce their outputs, we can execute parts of them in earlier cycles when the processor is available. Technically there are different methods for splitting the execution of the slow tasks to obtain a more balanced distribution of the computational effort.

Offline splitting

One approach consists in partitioning the code of every slow controller *offline* into pieces of approximately equal execution times and distributing their execution over all cycles inside its period. In our example this means splitting c_2 into c_{21} and c_{22}, splitting c_3 into c_{31}, c_{32} and c_{33} and using a cyclic executive to schedule the modified tasks, leading to a schedule like the one illustrated in Fig. 5. The corresponding schedulability condition becomes:

$$\max_j \sum_i C_{ij} < T.$$

This solution, which has many advantages, is quite popular in practice. For instance, it is the one adopted in the time-triggered architecture (TTA) framework [15], where it is handled by several commercial tools. One disadvantage of this approach is that the splitting of a control loop into subparts of similar execution time is not easy to accomplish at the application level (Simulink model) and possibly requires several iterations until a feasible schedule is

[3] Improvements can sometimes be achieved by using different phases (offsets) for the periodic computations.

found. Doing it directly on the code of the control program one loses some of the methodological advantages of automatic code generation. The variability in the execution times of the same program on modern processors does not make this job easier.

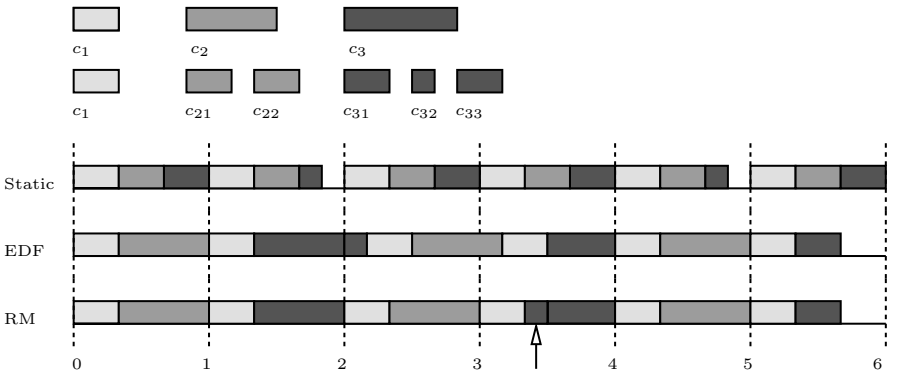


Fig. 6. Schedules for the S_{123} example: static splitting, EDF and RM

Preemptive solutions

The other class of solutions is more dynamic and is based on the preemption services of an RTOS. Every controller is compiled into a simple program, each instance of which is viewed as an independent task dispatched for execution by a scheduler according to some policy. The basic principle is that a slow process may execute when the computer is available, but when a more urgent task is released, the active computation is stopped and resumes when the urgent task terminates. This "context switching" (saving the contents of registers) takes some time, which we ignore in this discussion. The classical result of Liu and Layland [18] shows that, for preemptive scheduling, a set of tasks is schedulable if the amount of computation time to be consumed in P cycles is smaller than $P \cdot T$, that is,

$$\sum_i C_i/T_i < 1.$$

The two most popular scheduling policies are *earliest deadline first* (EDF) and *rate-monotonic* (RM).

Earliest deadline first: The simplest and most natural way to allocate the time budget of the processor is to prefer most urgent tasks: at any moment,

choose among the enabled tasks the one with the nearest deadline. If two or more tasks have the same deadline, an arbitrary choice can be made, with preference to tasks that are already executing (to minimize context switching). An example of an EDF schedule obtained for S_{123} appears in Fig. 6. Note that when the third instance of c_1 arrives, it does not preempt the first instance of c_3, because they have the same deadline. EDF was introduced in [18] and has been proven to be optimal.

Rate-monotonic: The alternative and rather popular approach is to use a *static* priority relation among tasks based on their frequency ($c_1 \prec c_2 \prec c_3$ in our case). Then at every time instant the task with the highest priority among the enabled ones is selected for execution. RM schedules tend to make many more preemptions than EDF and, even if we ignore context switching, they are provably less efficient than EDF schedules. As one can see in Fig. 6, S_{123} is not schedulable by RM on the same platform for which it is schedulable by EDF as the computation of the first instance of c_3 misses its deadline. The popularity of RM can be partly explained by the fact that fixed priority policies are easier to implement in existing operating systems, and that the degradation in performance with respect to EDF is only 1/3 in the worst case.

3.3 Semantic issues

The discussion in the previous section was based on a simplified abstract view of the controllers, assuming their I/O to be atomic operations that take place within zero time at the endpoints of each of their respective periods. We also implicitly assumed that the controllers are independent and do not communicate. In reality, the I/O operations are often part of the code of each task, and the timing of their execution may depend on the scheduling policy used. We mention two issues related to this fact: data consistency and determinism.

Data consistency

The first low-level problem to be resolved is due to the possibility that preemption occurs in the middle of an I/O operation, leading to corrupted data. For example, a task may be interrupted after having read some part of a long piece of data and resume operation only after some other task has modified it. Several solutions exist for this problem:

1. Protection by semaphores: This technique, used extensively in operating systems when resources are shared by several tasks, consists of preventing the interruption from occurring during I/O operations. From the point of view of priority-based scheduling this means that the task increases its priority when it enters its "critical section." This feature makes the scheduling problem more complex because the blocking time has to be evaluated and added to the WCET of the corresponding tasks. This can raise the

well-known *priority inversion* problem for which solutions such as the *priority inheritance* or *priority ceiling* protocols have been invented [24].

2. *Lock-free* methods: Here the reading task may detect the fact that the data it has been reading has changed and it may restart reading, attempting to get uncorrupted data [1, 16, 17]. Although the number of times this may happen is finite, the time that can be spent on retrying should be accounted for in the schedulability analysis.
3. *Wait-free* methods: Here data that are shared by several tasks are duplicated (double- or triple-buffers) so that the reader and the writer use different "lock-free" copies and than toggle between them. Consequently, the schedulability analysis need not be modified, but more space is needed to store the shared data [8, 14].

Determinism

Under this title we group all phenomena related to the deviation of the implementation from the "nominal" control loop that may result from the potential variability in execution times of different instances of the same task. We illustrate this class of problems and compare the influence of such variability on the three types of scheduling policies previously mentioned (simple, static splitting and preemptive). No attempt is made to cover the whole panorama of considerations and practical solutions.

Consider example S_{123} where controller c_1 has a state variable y_1 which is computed every iteration as $y_1' = f(y_1, y_2, y_3)$ where y_2 and y_3 are computed by c_2 and c_3, respectively (note that this also covers the special case where y_2 and y_3 are just inputs sampled at a lower frequency). Before continuing, it is worth contemplating the definition of the computed controller in terms of the *external time* of the controlled environment. If we were dealing with continuous time or with uniform sampling, the values of y_1, y_2 and y_3 used in every invocation of c_1 would be of the same real-time "age," that is, something of the form

$$y_1(t') = f(y_1(t), y_2(t), y_3(t)). \tag{2}$$

Since the y's are computed/sampled at different rates, each invocation of c_1 inside the super-period can use only the most recent values of y_2 and y_3 that are available, which leads to six different variations on (2), one for each cycle (see Fig. 7). For example, in the last cycle we compute $y_1(t) = f(y_1(t-1), y_2(t-2), y_3(t-3))$.

This "non-uniform" relation, expressed naturally using the under-sampling features of Simulink, is the starting point of multi-periodic control loops.[4] Under the simple scheduling policy, this relation is robust under variations in execution time because each task is executed in a predefined cycle. The

[4] In fact, the exact definition of this relation may vary according to the details of the I/O mechanism, but the important point is that the same pattern repeats every P cycles.

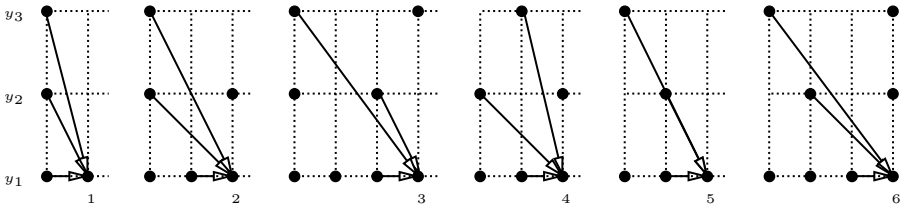


Fig. 7. Six different computations of $y_1' = f(y_1, y_2, y_3)$, each with a different external time characterization of the relation between the variables

situation is not much different if we use the static splitting approach, because the I/O operations appear in fixed portions of the code of each task, which are executed at predefined cycles.

On the other hand, preemptive methods are less robust in this sense as the I/O operations of a given instance of a task may occur at different cycles in different instances of the super-period depending on the point in the program where preemption takes place. For example, in the EDF schedule of Fig. 6, if c_3 takes less time and terminates within the second cycle, then the third invocation of c_1 may use this value, i.e., $y_3(t-1)$, instead of $y_3(t-3)$. A similar type of non-determinism, also known as *jitter*, is associated with the variation in the timing of the output operations. These types of non-determinism constitute one of the main criticisms of preemptive solutions for control applications. To alleviate this problem, various "time-triggered" solutions for the communication between different parts of the controller and for I/O in general have been proposed. Among them are the time-triggered architecture [15], to be discussed in Section 5, and the Giotto language [13] which allows preemption but isolates the execution of I/O operations from the rest of the code and forces them to take place in predefined time slots.

Let us remark that the attempts to maintain this determinism seem somewhat questionable, at least for periodic implementation of *continuous* control. The fact that the age of the value used by a controller deviates by a cycle or two between invocations need not have a significant effect on the performance of the control loop, given that such age variability already exists between consecutive cycles. Moreover, due to the measurements process and the variability of the external environment, there is not much sense in speaking of determinism in the actual execution of the control loop, although determinism is a convenient feature for debugging and simulation purposes. The situation may be different for a hybrid system where continuous and discrete-event control are combined (see Section 4.3).

4 Discrete Events and Hybrid Systems

So far we have focused on classical continuous control, whose implementation by computers is supported by the mature theories of sampled-data control and periodic scheduling. In this section we address the implementation of *discrete-event control* systems, which constitute an important ingredient of any modern control system and whose interaction with continuous control led to the emergence of a new field of research known as *hybrid systems*. Although such systems have been intensively studied in recent years, there is no comprehensive theory concerning their implementation, despite some recent efforts [5, 10].

4.1 Comparison with continuous control

The specification of a discrete-event controller is given in terms of a *transition system*, a generic term which covers automata, Petri nets or variants of Statecharts (state machines augmented with features such as parallelism and hierarchy). A transition system is defined over a discrete set of states and discrete sets of input and output events (alphabets). The dynamics is given in terms of a transition function consisting of tuples of the form (q, a, b, q') with the following intended meaning: when an input event a occurs while in state q, an output event b is generated and the controller moves to state q' (see Fig. 8). Note that the execution of the transition is not merely a table lookup operation as in textbook finite-state automata, but may involve manipulation of complex data structures which are part of the state of the system. The software implementation of a transition system is a program that decides according to the current state and the input event which reaction to compute.

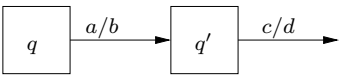


Fig. 8. A transition system

Although discrete-event systems are defined using the same abstract scheme of dynamic systems, that is, read input, update state and write output, their nature is quite different from that of continuous systems (see a more detailed discussion in [19]). In the latter, the dependence of the dynamics on the values of the state and the input is more or less continuous as these are variables appearing in the numerical recurrence equation. In discrete systems, the dynamics is defined by if-then-else statements where the values of state and input variables serve to choose among the branches of the program.

This leads to a much larger variability in the execution time for subsequent invocations of the controller.

The second major difference is associated with the time axis with respect to which the system is defined. The specification of continuous control systems is tightly and explicitly embedded in the real-time axis through the sampling rates which determine when inputs are read and what the deadline is for each invocation of a controller. Discrete transition systems are typically defined with no reference to real time and operate on a *logical time scale*, defined by the events. In other words, the model says that after input a there will be an output b, but any amount of time may separate the two events. The only implicit constraint is that the transition should be completed before the arrival of the next input event.

Without constraints on the environment, only an infinitely fast controller that reacts in zero time can respond to any event at any time. Assuming the existence of such a fast machine is often called the *synchrony hypothesis*, and it is advocated, among others, by the proponents of the Esterel language [4]. Although such machines do not exist, it is claimed that this *zero time* approximation provides reactive programming languages with a much cleaner and simpler semantics. As benefits, programs are easier to understand, debug and validate. Let us also note that this assumption is implicitly accepted during simulation, for example, with tools such as Simulink/Stateflow: each time the controller has an action to perform, the simulation time is frozen, and resumes only after the action is completed. Of course, stopping "real" time is much more difficult. We mention a recent variation on the synchrony hypothesis proposed in [21] where zero is replaced by a fixed and uniform delay (the *logical execution time*) in the semantics of the specification. The choice of this number, however, requires looking into the properties of the execution platform, except, perhaps, for systems where the reactions are very simple.

When moving to software implementations of such systems, we must bring real metric time into the picture, both at the specification level (refining the response time requirements, adding assumptions concerning the speed of the environment) and at the implementation level (execution times of the reactions on a given platform, event detection mechanisms). As no system can detect and handle events that arrive with an unbounded frequency, we need to convert the ideal "untimed" specification into a realistic one by adding constraints to the model of the environment so that such "Zeno behaviors" are prevented.

A simple and natural way to restrict the environment is to assume a positive lower bound on the inter-arrival time of events (events that violate this constraint are ignored by the controller). This is a very sensible requirement when the events are determined by changes in the values of discrete signals. An implementation of a system admitting such a lower bound d is guaranteed to meet the specifications if the WCET of each transition is smaller than d. Sometimes it is reasonable to assume such a lower bound for each type of

input event separately. This does not prevent an event of one type from arriving while the (sequential) implementation is busy reacting to another event. However, if the respective WCETs are small enough, the system can cope with these events using a *bounded buffer* that stores pending events (this is similar to the schedulability of multi-period systems discussed in Section 3).

Alternatively, one can explicitly set deadlines for each reaction or simply assign priorities so that the system will respond to the more important events and postpone the treatment of others while it is overloaded (this approach is common in "soft" real-time systems). As we have already noted, the determination of the real-time requirements is less systematic than in the case of continuous systems, and in many cases this part of the specification will be derived a posteriori from the constraints of the execution platform rather than in a top-down fashion.[5]

4.2 Implementation strategies

Let us illustrate two popular implementation styles without attempting to be exhaustive in the coverage of all existing approaches.

Single program time-triggered implementation

This is probably the most popular implementation strategy. It attempts to treat discrete-event systems using the same principles used for continuous ones. It is similar to the cyclic executive for multi-rate systems with which it can be easily combined, although no deep theory has been developed for it. We assume without loss of generality that events correspond to changes in values of Boolean signals. The set of controllers that specify the system is compiled into a single program, a sampling rate is chosen and it determines the deadline for the reactions to events. The input signals are sampled at a fixed rate and if a signal value is found to be different than in the previous sampling point, an event is declared. The reactions to all detected events are then executed sequentially and should terminate within the sampling period.

To see how this approach integrates easily with continuous control, consider, e.g., a train controller which must maintain a reference velocity using standard continuous control but which should react as well to events such as requests for emergency stops or other user commands. At every sampling point such a controller will read the continuous variables as well as the events. Then, it will execute the reaction for the events (some of which may cause mode switching in the continuous dynamics) followed by the computation of the continuous feedback function. Typically, no preemptive scheduling is used in this implementation style and no attempt is made to make efficient use

[5] In fact, this is also sometimes the case in continuous control where sampling rates are determined based on known limitations of the intended implementation platform.

of the computer. To be schedulable, the sum of WCETs of all the possible reactions (computed over the set of all input events that may occur within one sampling period) plus the WCET of the continuous control loop should be smaller than the sampling period.

Tasks and event-triggered implementation

Another popular implementation strategy starts with a collection of discrete controllers, each handling one class of events. Each controller is compiled into a *separate task* which is invoked when the event occurs. This approach requires using an RTOS and some scheduling policy: each event generates an interrupt and the scheduler decides whether to execute the corresponding task or wait for the termination of a task already being executed.

Fixed priority scheduling seems to be the most popular policy for this implementation style where, naturally, higher priority is assigned to tasks with closer deadlines (*deadline monotonic* policy). A nice schedulability analysis has been proposed in [3] for this policy under a minimum inter-arrival time condition. When such a condition holds, the approach does not suffer from the "unpredictability" charges that proponents of the time-triggered solutions tend to put on event-triggered systems [15].

The approach combines nicely with periodic and multi-periodic activations, for instance, by using a fixed priority preemptive scheduling policy for the periodic tasks. Actually, real-time clock activations can be seen as events among others, which are naturally endowed with a minimum inter-arrival time, the period itself. In this sense, this approach generalizes the multi-periodic one and is well adapted to hybrid systems.

Note that the two aspects mentioned, a single program versus separate tasks and periodic versus event-triggered sampling, are somewhat orthogonal. For example the implementation of a program written in the Esterel language is compiled into a single application task as in the time-triggered implementation. Then, this application task runs concurrently with another task, the *event handler*, which detects events and dispatches them for execution when the application task is idle.

4.3 Semantic issues

As we have noted in Section 3.3, variations in execution or communication time may cause changes in the external I/O behavior of controllers. In continuous systems this is restricted to the age of data used by the controller, but in discrete interacting systems the effect of such variations on the behavior of the controller can be more dramatic.

To illustrate this important phenomenon, consider the two automata appearing in Fig. 9 together with their composition. The first automaton reacts to a while the second reacts to b but its reaction depends on the state of the first. As one can see, the state of the system depends on the order in which

the two events arrive. In particular, according to the standard synchronous composition of automata, if a and b occur simultaneously, the outcome (for this example) is different than in the case where a occurs before b. Hence, in order to be faithful to the semantics of the model, the implementation should be able to make unrealistic distinctions. Only "confluent" automata admitting a diamond-like shape (as the one depicted on the right of Fig. 9) have their semantics robust under such variations, but such global automata are obtained only when the individual controllers are practically independent.

As an illustration, consider a periodic sampling implementation and the two signals of Fig. 10 whose respective risings generate the events a and b. A slight shift in the sample times may lead to two different interpretations: in the first a and b are perceived as occurring at the same time while in the second a occurs before b. How do designers take this phenomenon into account? It seems that they apply (consciously or not) tricks borrowed from the asynchronous hardware domain where such phenomena are called hazards or critical races.[6] For instance, they try to ensure that any possible race acts on independent state variables, and if this is not possible, they try to avoid the critical race by forbidding the inputs from changing at almost the same time. This, in turn, is obtained by imposing delays or causality relations between any two inputs that could possibly be involved in a critical race.

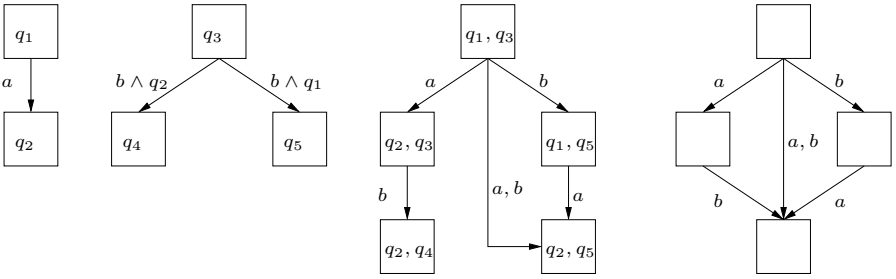


Fig. 9. Two interacting systems and their composition (the transition label a, b indicates that a and b occur simultaneously); a confluent automaton

For event-triggered preemptive implementations this problem is, of course, more severe, and several solutions for it have been proposed. As mentioned previously, the Giotto approach and its extension to discrete events [21] guarantee semantic consistency by deterministic timing of the I/O operations. On

[6] In many applications, software-based control has evolved from previous hardware implementations and the hardware culture is still vivid.

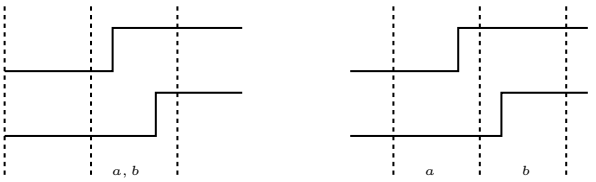


Fig. 10. A pair of signals interpreted differently depending on the sampling

the other hand, the solution proposed in [22] using a multi-buffer protocol achieves the same goal without insisting on timing determinism.

5 Distribution and Fault Tolerance

The preceding sections dealt with control systems implemented on a single computer. However, many large control applications are distributed for various reasons such as location of sensors and actuators, performance or fault tolerance. As a matter of fact, distribution and fault tolerance are strongly related issues: on one hand, fault tolerance usually requires some redundancy which can be implemented as distribution and, on the other hand, distribution raises consistency problems [20] that make fault tolerance more difficult to implement. For this reason we treat them in the same section, which is somewhat superficial, given the huge amount of work dedicated to distributed computing during the past thirty years. We simply mention the major problems and discuss briefly two classes of solutions used in control applications.

A distributed platform consists of several computers, sensors and actuators (nodes) linked together via some communication network through which data can be transmitted. An implementation of a control system on such an architecture consists of assigning controllers to processors, scheduling them and specifying the communication protocol according to which different nodes in the network interact. This architecture aggravates the semantic problems associated with a single computer implementation, namely, variability in execution times and ordering of events, due to communication delays, clock drifts between different processors, etc.

5.1 Local clocks solutions

This is the most widely adopted solution in distributed control systems up to now. The idea is quite simple:

- Each computer has a local real-time clock and runs a periodic (or multi-periodic) application as described in Section 3.

- Each computer samples its external world periodically based on its local clock. This world is made of its physical environment and variables produced by other computers. This amounts to a shared buffer (shared memory) inter-computer communication mechanism.

This solution has many advantages as each computer is complete and acts autonomously. This feature matches pretty well modern aspects of computation and control, as manifested in sensor networks and Internet-based control. The implementation does not require specialized hardware and can thus take advantage of the fast performance improvements and world-wide debugging of mass market products.

Yet, this approach has several drawbacks. Due to the lack of clock synchronization, it yields large jitters that may become larger than the periods. For a purely continuous system this problem is not so severe, because the deviation in the real-time age of data items is always bounded. However, it can become more serious when discrete events are involved. Another drawback is that when two systems are not synchronized, they should observe each other more frequently in order not to miss events.

As shown in [7], redundancies can be implemented on top of such systems in order to achieve fault tolerance.

5.2 Global clock solutions

These are emerging solutions which have been subject to a large research effort in the past years. They are best known as *time-triggered* solutions [15, 23] and are based on the following principles:

- A redundant bus dispatches a common fault-tolerant real-time clock to each computer.
- Communication between computers takes place at fixed points in time determined by the global clock.
- Each computer runs a periodic or non-preemptive multi-periodic (see Section 3.2) application driven by the global clock.

The major advantage of this solution is that it yields small jitters as the timing is very deterministic. It comes equipped with built-in fault-tolerance strategies and with toolboxes integrated with Simulink/Stateflow which alleviate the transition from models to implementation. The drawbacks are exactly the opposite of the advantages noted in Section 5.1: the approach is less flexible and may be more expensive and less efficient as it requires specialized hardware.

As a matter of fact, these two solutions can be seen more as complementary rather than competing. The local clock solution is well adapted to *loosely coupled* (autonomous, asynchronous) systems while the global clock solution matches *tightly coupled* ones. Moreover, in control systems distributed over

large distances, there will always be subsystems that are not synchronized and will need the local clock solution.

Another striking fact about this landscape is that both solutions are time triggered. It seems as if the event-triggered option has not been considered for control-dominated distributed systems, while it is the dominant approach for most distributed systems oriented toward communication and computing. This could be a topic for future research, especially as control and communication become more and more interdependent.

References

1. J. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. In *Proceedings of the 16th Real-Time Systems Symposium*, pages 28–37. IEEE Computer Society, 1995.
2. K. J. Åström and B. Wittenmark. *Computer Controlled Systems — Theory and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1996.
3. N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. Hard Real-Time Scheduling: The Deadline Monotonic Approach. In *Proceedings 8th IEEE Workshop on Real-Time Operating Systems and Software*, Atlanta, GA, 1991.
4. G. Berry and G. Gonthier. The ESTEREL synchronous programming language, design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.
5. P. Caspi and A. Benveniste. Toward an approximation theory for computerised control. In A. Sangiovanni-Vincentelli and J. Sifakis, editors, *2nd International Workshop on Embedded Software, EMSOFT02*, volume 2491 of *Lecture Notes in Computer Science*, pages 294–304, 2002.
6. P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating discrete-time Simulink to Lustre. In R. Alur and I. Lee, editors, *3rd International Conference on Embedded Software, EMSOFT03*, volume 2855 of *Lecture Notes in Computer Science*, pages 84–99, 2003.
7. P. Caspi and R. Salem. Threshold and bounded-delay voting in critical control systems. In Mathai Joseph, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems*, volume 1926 of *Lecture Notes in Computer Science*, pages 68–81, 2000.
8. J. Chen and A. Burns. Loop-free asynchronous data sharing in multiprocessor real-time systems based on timing properties. In *Proceedings of the Real-Time Computing Systems and Applications Conference*, pages 236–246, 1999.
9. J.-L. Colaço and M. Pouzet. Clocks as first class abstract types. In R. Alur and I. Lee, editors, *Third International Conference on Embedded Software (EMSOFT'03)*, volume 2855 of *Lecture Notes In Computer Science*, pages 134–155, 2003.
10. V. Gupta, T. A. Henzinger, and R. Jagadeesan. Robust timed automata. In O. Maler, editor, *Hybrid and Real-Time Systems, HART'97*, volume 1201 of *Lecture Notes in Computer Science*, pages 331–345, 1997.
11. N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

12. D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems*, volume 13 of NATO *ASI Series*, pages 477–498. Springer-Verlag, Berlin, 1985.
13. T. A. Henzinger, B. Horowitz, and Ch. M. Kirsch. Giotto: A time-triggered language for embedded programming. *Proceedings of the IEEE*, 91:84–99, 2003.
14. H. Huang, P. Pillai, and K. G. Shin. Improving wait-free algorithms for interprocess communication in embedded real-time systems. In *USENIX 2002 Annual Technical Conference*, pages 303–316. http://www.usenix.org/publications/library/proceedings/, 2002.
15. H. Kopetz. *Real-Time Systems Design Principles for Distributed Embedded Applications*. Kluwer, Dordrecht, 1997.
16. H. Kopetz and J. Reisinger. Nbw: A non-blocking write protocol for task communication in real-time systems. In *Proceedings of the 14th Real-Time System Symposium*, pages 131–137. IEEE Computer Society, 1993.
17. L. Lamport. Concurrent reading and writing. *Communications of ACM*, 20(11):806–811, 1977.
18. C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the Association for Computing Machinery*, 20(1):46–61, 1973.
19. O. Maler. Control from computer science. *Annual Reviews in Control*, 26:175–187, 2002.
20. M. Pease, R. E. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–237, 1980.
21. M. A. A. Sanvido, A. Ghosal, and T. A. Henzinger. xgiotto Language Report. Technical Report UCB/CSD-3-1261, Computer Science Division (EECS) University of California Berkeley, July 2003.
22. N. Scaife and P. Caspi. Integrating model-based design and preemptive scheduling in mixed time- and event-triggered systems. In *Euromicro Conference on Real-Time Systems (ECRTS'04)*, Catania, June 2004. IEEE Computer Society.
23. C. Scheidler, G. Heiner, R. Sasse, E. Fuchs, H. Kopetz, and C. Temple. Time-Triggered Architecture (TTA). In *Proceedings EMMSEC'97*, Florence, Italy, November, 1997.
24. L. Sha, R. Rajkumar, and J. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.
25. R. Wilhelm. Determining bounds on execution times. *Handbook on Embedded Systems*, CRC Press, Boca Raton FL, 2005.

Embedded Real-Time Control via MATLAB, Simulink, and xPC Target

Pieter J. Mosterman,[1] Sameer Prabhu,[2] Andrew Dowd,[3] John Glass,[1] Tom Erkkinen,[4] John Kluza[5], and Rohit Shenoy[6]

[1] The MathWorks, Inc., Simulink Development, 3 Apple Hill Dr., Natick, MA 01760-2098, U.S.A.
[2] The MathWorks, Inc., Application Engineering, 39555 Orchard Hill Place, Novi, MI 48375-5374, U.S.A.
[3] The MathWorks, Inc., xPC Target Development, 3 Apple Hill Dr., Natick, MA 01760-2098, U.S.A.
[4] The MathWorks, Inc., Technical Marketing, 39555 Orchard Hill Place, Novi, MI 48375-5374, U.S.A.
[5] The MathWorks, Inc., Application Engineering, 3 Apple Hill Dr., Natick, MA 01760-2098, U.S.A.
[6] The MathWorks, Inc., Technical Marketing, 3 Apple Hill Dr., Natick, MA 01760-2098, U.S.A.

1 Introduction

This article shows how xPC Target [44] facilitates embedded control system design by turning general-purpose personal computer (PC) hardware into a rapid prototyping platform. The PC-based platform used is the MathWorks xPC TargetBox™ [45], an industrial PC. xPC Target is integrated in Simulink® [31], enabling the use of Simulink as a graphical front end with MathWorks tools for parameter estimation, response optimization, and linearization throughout the design cycle.

1.1 What is an embedded control system?

A control system is an implemented strategy used to cause a physical system, or *plant*, to behave in a desired manner. There are two types of control strategies:

- Closed-loop control uses feedback measurements to correct error between the plant output and a reference input, i.e., the desired behavior.
- Reactive control is event driven and interacts with the plant via state transition behavior.

As the feedback control strategy increases in complexity, it becomes more difficult to apply analog components for its implementation. Dynamics in

an analog feedback control loop always interact, making it more difficult to match desired controller characteristics. For example, an analog system always has a limited filter quality factor, Q, due to parasitic impedances and other limitations. Conversely, it is easy to create an extremely sharp digital filter with very large Q. Another complication is that analog integrators are always limited by capacitor leakage, yet digital integrators can be nearly perfect.

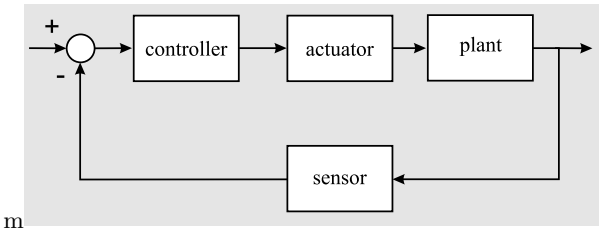
A processor-based approach usually works best for reactive control as well.

In modern control systems, the control strategy is thus typically implemented in software. A microprocessor determines the input to manipulate the plant and this requires facilities to apply this input to the physical world. In addition, the control strategy typically relies on measured values of the plant behavior that have to be made available to the computing resources.

The immersion of computing power into the physical world is one characteristic of an *embedded control system*. The other characteristic is that the software that implements the control strategy is stored in read-only memory. Thus, unlike a general-purpose computer, an embedded control system is not independently programmable. In other words, an embedded control system is expected to function without user *intervention*, although it may require user *interaction*.

1.2 Embedded control system characteristics

The general configuration of an embedded control system is shown in Fig. 1. Because the controller operates in the low-power electronics domain and the plant operates in high-power hydraulics, mechanics, thermal, and other physical domains, transducers are needed to convert between controller and plant. These transducers are used either by *actuators*, to drive the plant with controller-computed values, or by *sensors*, to provide measurements to the low-power electronics domain. In embedded systems, the low-power computational electronics of the controller has to interact with high-power physical domains of many types [42].



m

Fig. 1. General embedded control system configuration

For example, consider the Stewart platform in Fig. 2. This physical system consists of six legs supporting a circular platform. The platform may be used

to build, for example, an aircraft simulator. The legs are then used to move the simulated aircraft so as to give the impression of being inside an actual aircraft. This unit is sometimes called a "hexapod," after its six legs.

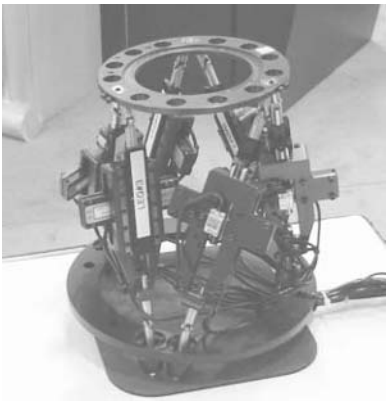


Fig. 2. A Stewart platform

To move the platform, each leg is equipped with a motor that extends it. The control strategy that computes the desired extension is implemented in a low-power microprocessor. An amplifier turns this electrical signal into a high-power equivalent that can be used to drive the motor. Sensors measure the actual extension of the legs. Six linear encoders, one on each leg, send a voltage pulse every time the leg slides a given distance. Dedicated counter hardware counts the number of pulses. The actual distance is computed based on this count.

In addition to the transformation between high- and low-power domains, transformations between discrete-time and continuous-time behavior are required. The plant can be viewed as changing continuously in time [14, 27]. The controller, however, has a discrete clock that governs its behavior, and so its values change only at discrete points in time. To obtain deterministic behavior and ensure data integrity, the sensors must include a mechanism to sample continuous data at discrete points in time, while the actuators need to produce a continuous value between the time points with discrete-time data (typically, the value is held constant).

1.3 Rapid prototyping in embedded control system design

Formal control design methods invariably rely on a plant model [1, 4]. The plant model can be derived from first principles but often contains unknown parameters. Experiments must be conducted to gather information on the behavior of the plant dynamics to help estimate these parameters.

Once a plant model is available, closed-loop feedback and reactive control can be designed using simulation or synthesized using methods such as pole placement, inverse dynamics, total energy control, H_∞, and model predictive control.

Rapid prototyping tools support this design paradigm. At the start of the control design process an engineer may have an inaccurate model or no model at all. At this stage, a skeleton control system is developed to stabilize a system and to obtain the desired behavior for the experiment. Experiments can then be designed and performed to acquire responses of the system under various operating conditions. The acquired data can then be used to enhance the plant model and to design a new control system based on the more accurate plant model. Simulating the combined control system and plant model, the designer can study and optimize the performance of the system using the full nonlinear plant simulation model. Finally, the control system can be implemented on a rapid prototyping system. If the system does not meet the performance achieved in simulation, the model and the control system design are further refined.

Such incremental design for embedded control systems requires that the rapid prototype operate in real time, interact with hardware, have supporting control functionality, and be safe.

1.4 Chapter overview

In Section 2, the concept of rapid prototyping is elaborated. In Section 3, the Stewart platform is presented that will be used throughout the chapter to illustrate the concepts put forward. Section 4 discusses the PC-based xPC Target for rapid prototyping, and Section 5 describes the industrial xPC TargetBox that is used to implement the embedded control for the Stewart platform. In Section 6, generation of the embedded code for control is discussed. Section 7 discusses how models are obtained. Section 8 explains how to acquire the data necessary for modeling. Section 9 gives an overview of how models are used in the embedded control system design. Section 10 describes the control strategy as used for the Stewart platform, and Section 11 presents conclusions.

2 What Is Rapid Prototyping?

Much research has been devoted to the analysis, design, and synthesis of a controller based on a plant model. Note that this research pertains to a model of the *controller* as well. Once this controller model has been designed, however, it still has to be realized and connected to the actual plant, and most of the actual control system engineering effort is devoted to taking the controller model to such a realization. In particular, accounting for some of the implementation details such as, for example, the resolution of a fixed-point

microprocessor that will be used, may affect the originally designed controller model and require it to be modified [20].

A rapid prototype is a quick way to validate the controller code by executing it with the actual plant, sensors and actuators, the plant model, or any combination of these components [19].

The purpose of rapid prototyping is to obtain confidence and pinpoint flaws and errors in a partial design before committing to a completed design. This is a common design approach. For example, in software, a core algorithm is typically implemented and tested before extensive comments, exception handling, and robustness functionality are added.

In scientific research and education, where a system is rarely taken into production, rapid prototyping serves an important purpose. In industry, rapid prototyping allows testing of a partial design before expending the effort to include robustness measures and optimizing the design.

There are three different rapid prototyping configurations: functional, bypass, and on-target (Fig. 3).

- *Functional rapid prototyping* is used for testing new ideas and research projects where there is no controller or the controller is too primitive to support advanced control strategies. In such cases, the rapid prototyping controller controls the entire system. As the focus is on proving the concept, the size of the generated code and the fixed-point characteristics of the software are not important. The hardware used for functional rapid prototyping is often PC-type hardware and is not intended for production controller applications. The flexibility to add I/O hardware is important in rapid prototyping as various hardware contingencies cannot be accounted for ahead of time.
- *Bypass rapid prototyping* replaces only a part of the existing control system with the new controller. This is useful if the system control demands high current capacity drivers that are typically not available in rapid prototyping controllers or if only part of the functionality of the controller needs to be replaced with the new features.
- *On-target rapid prototyping* uses the production hardware directly and captures all the hardware dependencies and I/O limitations [5,12]. It allows engineers to assess the ability of the algorithm to control a vehicle under various test track conditions, especially those, such as ice, that are hard to simulate using a model.

3 A Stewart Platform

This example describes the rapid prototyping of the Stewart platform shown in Fig. 2, using xPC Target [44] and xPC TargetBox [45].

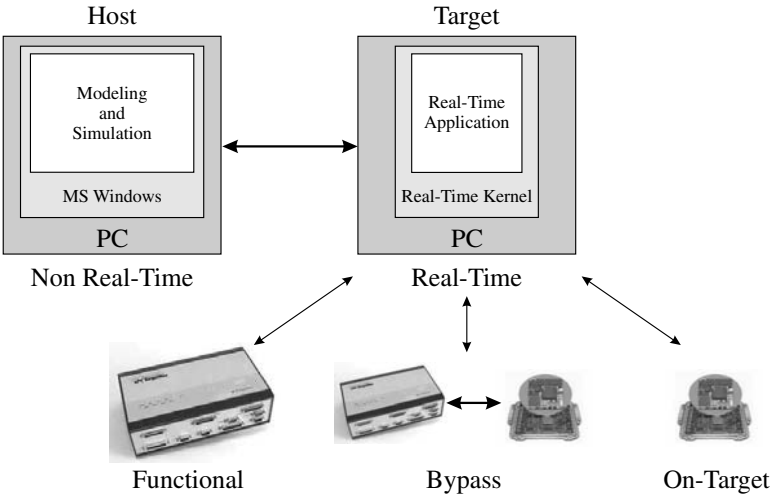


Fig. 3. Rapid prototyping configurations

3.1 Control objectives

One control objective is to enable the Stewart platform to assume a prescribed position as accurately and quickly as possible. Another is to move the platform at specific speeds.

3.2 System configuration

The Stewart platform system in Fig. 4 shows the hexapod plant and xPC TargetBox controller connected by sensors and actuators. xPC TargetBox includes a 400 MHz Intel Pentium III (floating-point) processor, with 128 MB RAM, and 32 MB flash RAM.

3.3 The peripherals

The peripheral hardware consists of force actuators and position sensors.

Force actuators

Mounted on the legs of the Stewart platform are Nanomotion H1 piezoceramic motors that extend the legs, so there are six actuators. The motors are driven by Nanomotion amplifiers that read the control voltage from one of the six channels of the RTD DM6604 analog output of the xPC TargetBox. The input to the amplifiers is a low-power analog reference signal that they convert into a high-power sinusoidal voltage. By varying the amplitude of the sinusoid, the motor moves the leg up and down.[7]

[7] http://www.nanomotion.com/data/docs/Tech%20notes%20102.pdf

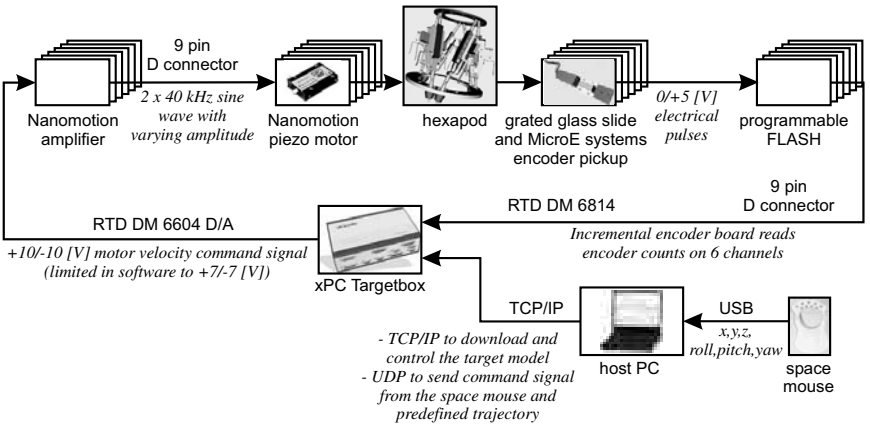


Fig. 4. Stewart platform hardware configuration

Position sensors

Six MicroE Mercury 3110 incremental encoders[8] measure the extension of each of the six legs. To measure the extension, these sensors use an optical beam that produces a sequence of electrical pulses when a grated slider is passed by. The slider has 65,536 counts on it over about 4 cm of travel, giving a precision of about 0.61 μm of travel per count.

A slide with a reference marking calibrates the zero location from which to start counting incrementally. In this particular hardware setup, the encoder pickups cannot see the reference marking and so have a purely incremental capability. Because the encoder cannot be reset at a given location based on a reference reset pulse, the only option is to drive it to the stops of the actuator slider and define that to be zero.

The sensor is powered by a 5 V supply at 300 mA. It delivers the electrical pulses with a power and impedance that allow it to be directly connected to a counter board. xPC TargetBox includes an RTD DM6814 incremental encoder board that counts how many pulses it receives from the encoder pickup and passes this count to the model. Two counter I/O boards, each supporting three channels, are used.

4 xPC Target

4.1 General-purpose hardware

A rapid prototyping platform needs to be more powerful and flexible than the eventual target processor. For example, if the software has not yet been

[8] http://www.microesys.com/

optimized, it will not run as efficiently. To achieve real-time behavior, a more powerful microprocessor is necessary. Furthermore, additional measurements may need to be made to obtain insight in the functioning of the controller.

The necessary flexibility, computing power, and memory capacity may make rapid prototyping platforms much more expensive than the hardware that is ultimately used in production. Because of the cost, rapid prototyping platforms are often used for more than one project, an approach that is supported by the inherent flexible nature of such platforms.

xPC Target [44] provides the means to turn general-purpose PC hardware into a prototyping environment that can be used for signal acquisition, rapid prototyping, and hardware-in-the-loop simulation.

4.2 PC form factor

The form factor of a device is its physical shape and size. There are a number of specific form factors available for PC-based systems. A form factor may encompass design components such as connector types, bus protocols, board sizes, power specifications, and mechanical enclosures. Rarely does the form factor directly influence processor selection, but a particular form factor is often indirectly tied to a processor family. Thus, it is common to couple the choice of a form factor to the processor selected. This can be a regular desktop PC, rack-mounted PC, or an industrial PC. The advantages of using PC-based platforms are their scalable computing power, flexibility, and expandability.

xPC Target can be used with any PC containing Intel 386/486, Pentium, or AMD K5/K6/Athlon processors as the real-time target. This includes desktop computers, industrial computers such as xPC TargetBox, PC/104, PC/104+, CompactPCI, all-in-one embedded PC, or any other PC-compatible form factor. Thanks to economies of scale and competition, these devices have performances in the order of millions of floating-point operations per second (MFLOPS), relative to cost. Moreover, the large range of available form factors allows xPC Target to be used in small PC/104 systems as well as in much larger expanded PCI systems. For example, it is common to perform early design work using a standard desktop PC and then immediately retarget the control algorithm to an industrial computer for field testing.

4.3 Real-time operating system

The xPC Target kernel provides a real-time operating system that supports both interrupt handling and polling and is tuned to provide maximum performance with minimal overhead. High-performance hardware allows sample rates that approach 100 kHz.

4.4 Drivers

A key step in transforming software into a real-time system is the requirement to have device drivers that communicate between the I/O devices on the target

PC and the application code running on this target. These drivers thus enable interaction between the real-time application and the real physical system. The device driver contains the code that runs on the target hardware for interfacing to I/O devices such as analog-to-digital (A/D) converters, encoders, digital signals, and communication ports. Each device driver is implemented as a Simulink S-function using C-code MEX files.

Fig. 5 shows the Simulink blocks for the six encoder channels supported by two DM6814 boards used on the Stewart platform. These blocks result in automatically generated code for the hardware drivers.

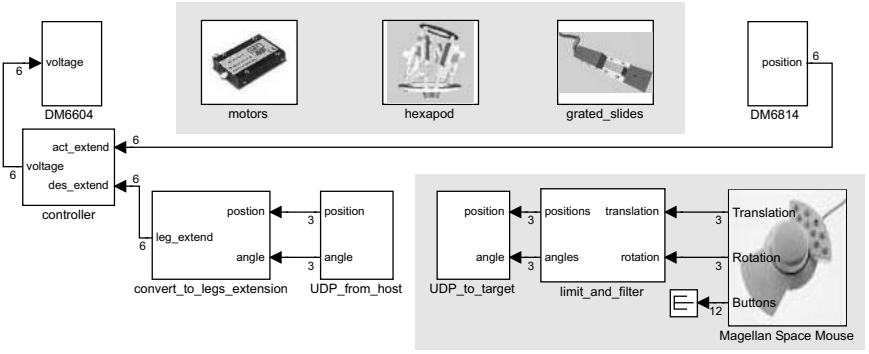


Fig. 5. Simulink model of Stewart platform host and target software

The code for the entire system identification application can be generated without manually producing glue or driver software.

4.5 Writing device drivers

To understand the process of writing device drivers, it is essential to understand S-functions and low-level programming of I/O boards.

An S-function is a description of a Simulink block written in a language such as M or C [31]. S-functions have a special calling syntax, referred to as a call-back, that allows these custom blocks to interact with Simulink in the same manner as built-in Simulink blocks do. A C S-function can be compiled and dynamically linked into the Simulink environment, thereby allowing custom blocks to be added to the Simulink environment. Thus, S-functions and S-function routines form an application program interface (API) that allows the flexible implementation of generic algorithms within the Simulink environment. This flexibility cannot always be maintained when S-functions are used with Real-Time Workshop® [28] to generate code. For example, it is not possible to access the MATLAB® [18] workspace from an S-function that is used with Real-Time Workshop, but it is possible when using the S-function with Simulink for simulation purposes only.

To incorporate a device driver block in the Simulink model requires an S-function, which in turn requires the C source code for the device driver. xPC Target provides a comprehensive device driver library supporting more than 250 boards of various types, including A/D converters, digital input/output, Controller Area Network (CAN) [3], and ARINC 429. Thus, xPC Target greatly simplifies the process of generating a real-time application by providing the library of device driver S-functions.

Writing a device driver that is not yet available is simpler for xPC Target than writing a general configuration since xPC Target does not contain the many layers typically found in an operating system (OS). For example, the xPC Target kernel has direct virtual-to-physical address mapping, which means that declaring a pointer at a particular address will lead to bus access at the same physical address. Moreover, the source code for the existing xPC Target device drivers is provided with the product, enabling users to gain familiarity with the way device drivers are implemented.

Device drivers for xPC Target can be developed in one of two ways:

- Obtaining the source code for the driver from the hardware manufacturer and porting it to the xPC Target kernel
- Using the register programming manual of the I/O board from the hardware manufacturer to develop a driver from the very beginning.

The xPC Target kernel provides a set of functions for accessing ports and memory, PCI initialization space, and performing time measurements that can be used in an S-function compiled for xPC Target.

PCI boards are better than ISA boards for adding I/O functionality to a real-time system because they provide plug-and-play allocation of access, interrupt resources, a wider and faster bus, and software calibration.

4.6 xPC Target configuration

The xPC Target host-target arrangement is shown schematically in Fig. 6. On the host PC (which runs MATLAB, Simulink, Real-Time Workshop, and xPC Target), xPC Target works with the code generated from the Simulink application and a C compiler to build the real-time target application. The target application can run in real time on a target PC once it is downloaded to the target PC from the host PC. The target hardware is booted from a real-time kernel in xPC Target. However, the xPC Target kernel needs the PC basic input/output system (BIOS) because when the target PC boots and the BIOS is loaded, the BIOS prepares the target PC environment for running the kernel and then starts the kernel.

The kernel initiates the host-target communication, activates the application loader, and waits for the target application to be downloaded from the host PC. The host-target communication can occur through either serial or TCP/IP communication protocols. Once the target application has been

downloaded to the target PC, it can be controlled and modified from the host PC.

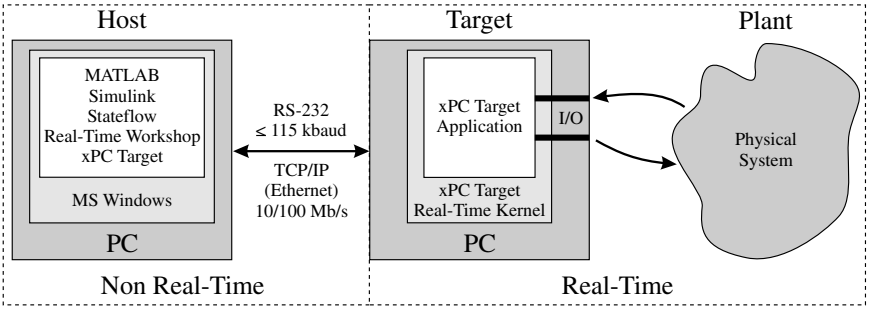


Fig. 6. xPC Target system

In the annotated Simulink model of the Stewart platform setup shown in Fig. 5, the shaded area in the bottom-right corner marks the software that is running on the host PC. The shaded area at the top indicates the actual hardware. The rest of the blocks are running on the target PC.

4.7 Host and target interaction

It is frequently necessary to interact with the real-time application to either observe signals or change parameters.

Data logging and on-line monitoring

During the rapid prototyping stage, it is important to have access to many variables. For this reason, the plant is typically instrumented with more sensors than will be included in the "production" configuration. In addition, the data needs to be stored in a persistent form or made visible in real time.

Oscilloscopes such as the Agilent 54621A 2-Channel 60 MHz Oscilloscope allow communication over, for example, an RS-232 connection [39]. This connection supports sending commands to the oscilloscope such as the time base to use. It also supports communicating the display data.

Alternatively, monitoring software can be used to manage data. xPC Target supports several monitoring and data logging methods, including xPC Target scopes, outport blocks in the Simulink model of the target application, and a Web browser interface.

xPC Target scopes are data display options for the target. Any signal in the target application can be associated with the target scope. In addition to being displayed on the display monitor attached to the target, the data that is sent to the scope can be stored in RAM or on the xPC Target file system and transferred to the host when the execution of the target application stops.

The availability of a file system on the target hardware allows large amounts of data to be logged, especially useful in prototyping applications. The data display and logging can be controlled by other signals in the model so that bursts of logged data can be acquired.

Outport blocks in the Simulink model of the target application can be used to log data to an object in the MATLAB workspace once the execution of the target application is terminated. From here, the data can be manipulated as regular workspace variables, one option being saving it to a file and another being displaying it in a MATLAB plot. The outport blocks must be at the top level in the model hierarchy and are considered model output. Time and the model state can be logged in the same way in which this data can be written to the MATLAB workspace during operation of the target application.

A Web browser interface can be used to retrieve the data logged by the target application in a comma-separated list that can be easily handled by spreadsheet or similar programs.

Note that the task execution time is a variable that is available for logging by an xPC Target real-time application, although it is not available when simulating the application in Simulink.

For the Stewart platform described in Section 3, the sample rate of the input and output blocks is chosen at 1 ms, yielding a data acquisition frequency of 1 kHz. This frequency is fairly standard for mechanical systems. Because it is much higher than the mechanical dynamics (around 10 or 20 Hz), it is high enough to eliminate aliasing concerns [8, 23]. The powerful xPC TargetBox processors and memory allow sampling at this high a frequency. To prevent data files from becoming too large, the data logging frequency is down-sampled by a factor of 10 to about 100 Hz.

Parameter tuning

xPC Target supports the modification of parameters in the Simulink blocks while the application is running. The parameter changes are immediately reflected in the real-time application. The tight integration between MATLAB, Simulink, Real-Time Workshop, and xPC Target makes it possible to write a script that incrementally changes a parameter and monitors a signal output. The script can then be run on the host PC to optimize the value of the parameter.

5 xPC TargetBox

xPC TargetBox provides a complete hardware capability for prototyping control systems. It combines xPC Target software with a Pentium-based computer in a rugged enclosure that is suitable for industrial environments. The microprocessor can be augmented with a number of I/O configurations that are commonly required for control applications, such as counters/timers, A/D,

D/A, pulse-width modulation (PWM), digital I/O, and CAN bus. xPC TargetBox is a PC-compatible computer configured to use a standard PC/104 stack but with all physical considerations incorporated to create a rugged and powerful controller.

xPC TargetBox includes chassis, enclosure, connector breakouts, and an internal power supply. It enables the design of embedded systems for applications such as mobile controllers like PC/104 and single-board computers (SBCs). The acquisition cost for an all-in-one embedded PC is slightly higher than for a PC/104 or SBC system, but there is no additional cost for designing and manufacturing an enclosure because the system includes the enclosure.

xPC TargetBox systems can achieve sample rates approaching 60 kHz. They accommodate up to four PC/104 expansion boards and support a selection of commonly used I/O options.

6 Generating Embedded Code

6.1 Application execution

Simulink simulation steps

A typical Simulink block consists of inputs, states, and outputs, where the outputs are a function of the sample time, the inputs, and the block states. During simulation, the model execution follows a series of steps (Fig. 7). The first step is the initialization of the model, where Simulink incorporates library blocks into the model; propagates signal widths, data types, and sample times; evaluates block parameters; determines block execution order; and allocates memory. Simulink then enters a simulation loop. Each pass through the loop is referred to as a *simulation step*. During each simulation step, Simulink executes each of the model blocks in the order determined during initialization. For each block, Simulink invokes functions that compute the values of the block states, the derivatives, and the outputs for the current sample time. The simulation is then incremented to the next step. This process continues until the simulation is stopped.

Real-time execution

Real-time behavior is inherent to embedded systems design. There are different definitions of "real-time". For the purpose of this paper, it is defined as "a fast enough response" for a particular application.

Real-Time Workshop takes the Simulink model and generates the application or algorithm code that contains the system of equations derived from the model as well as the block parameters and the code to perform initialization. Real-Time Workshop also provides a run-time interface that allows the model code to be built into a complete, stand-alone program that can be compiled and executed. Fig. 8 provides a high-level view of the real-time executable.

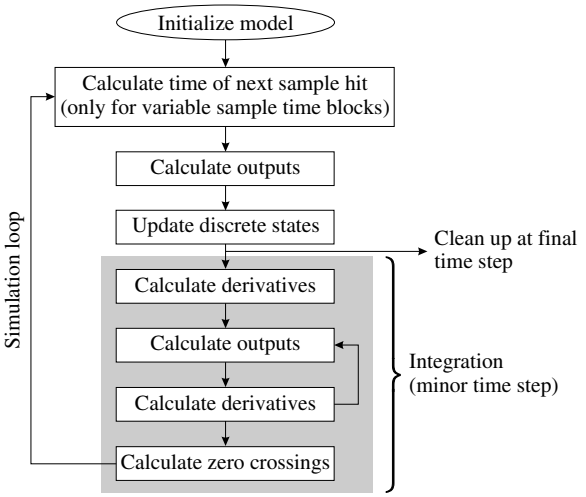


Fig. 7. Steps in a Simulink simulation

xPC Target uses this run-time interface and combines it with a real-time clock and scheduler to generate a real-time application, while providing the drivers for interfacing to real-time hardware and the signal monitoring and parameter tuning capabilities. Based on the sample rate specified in the Simulink model, xPC Target uses interrupts to step the execution of the model at the appropriate rate. With each new interrupt, the target application computes all the block outputs from the model, similar to the way Simulink computes its block outputs.

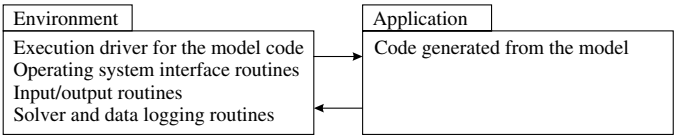


Fig. 8. The object-oriented view of a real-time program

The code generated from the Simulink model is sometimes referred to as the model code because it implements the Simulink model. The model code contains functions that correspond to the applicable simulation steps outlined in Fig. 7: compute the model outputs, update the discrete states, integrate the continuous states (if applicable), and update time. xPC Target generates its own main program. This program interacts with the execution driver for the model code, which in turn calls these functions.

The functions then write their calculated data to the real-time model. At each sample interval, the main program passes control to the model execution function, which executes one step through the model. This step reads inputs

from the external hardware, calculates the model outputs, writes outputs to the external hardware, and then updates the states, as shown in Fig. 9.

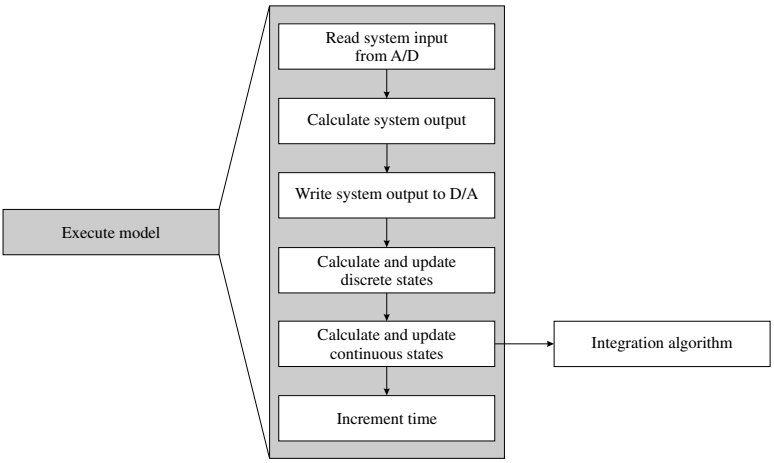


Fig. 9. Real-time execution of the model code

If these computations require the plant output of the previous sample time, they must be performed in one sample interval. This implies synchronization between the execution of the logical program by the controller and the dynamic behavior of the plant in real time. The sample rate is determined by control law analysis and depends on the time constants of the plant: the faster the plant time constants, the higher the required sample rate.

Note that this scheme writes the system outputs to the hardware before the states are updated. Separating the state update from the output calculation minimizes the time between the input and output operations. The generated code also contains functions to perform initialization, facilitate data access, and complete tasks before program termination.

The requirement to have plant input computed at a given point in time implies a fixed controller response time. As a result, the controller cannot rely on iterative computational schemes unless the upper bound of the iterations is fixed. This means that the controller must not employ a variable integration step or include algebraic loops.

6.2 Model-based code

Using Simulink as a graphical front end to the embedded software combined with automatic code generation technology makes it easy to modify the controller —it is easier to change the model than to change the code (code changes have a higher probability of introducing new defects) [22]. The controller can

be analyzed in terms of the Simulink model, which is more intuitive than the embedded software code, and sophisticated data analysis tools are immediately available to study and tune the controller performance.

6.3 Data acquisition code

As illustrated by the measurement setup in Section 3, the actuation of a plant and sensing some of its signals can be an intricate matter. The transducers used to transform signals between physical domains are often complex devices with highly nonlinear characteristics, making them difficult to model. Furthermore, careful calibration is crucial and, given that the physics of the system change over time, conscientious recalibration is a necessity.

Selecting from among the many available sensors and actuators is an important stage in the design of an embedded control system, especially because dedicated "signal conditioning" hardware may be required to employ particular sensors and actuators. This hardware may be used, for example, to change the impedance of a signal, ensure its voltage range is within required bounds (often between 0 V and 5 V), filter voltage spikes, and protect against power surges.

In addition, the actuators and sensors chosen present interfacing requirements. For example, a DC motor may have to be driven by PWM, and so the availability of a PWM channel to the controller is desirable.

A measured variable is made available to the embedded controller as a voltage. To be used in control law computations, this variable must be converted into the corresponding value of the physical quantity that it measured. For example, the position measurement of the legs of the Stewart platform is made available as a sequence of electrical pulses. These pulses are counted. The resulting value is indicative of the extension. However, the actual value requires computing $0.04/65536 * count\_number$ (see Section 3.3) first. Embedded control systems include software to perform the computations required to complete the data acquisition.

6.4 Supporting control algorithms

Embedded control systems typically account for different operating regions, or *modes of operation*. For example, an aircraft moves through a sequence of modes such as take-off, cruise, descent, and flare on each trip. In order to test the algorithm for a particular mode, the plant must be in the corresponding mode. It may be necessary to implement control algorithms for any of the modes that the system has to move through to arrive at the desired mode. Furthermore, additional control loops may be present in the same system. These supporting control algorithms can, however, be of a rudimentary nature.

To estimate friction parameters in the Stewart platform in Fig. 2, the system has to be moved to an operating point. This involves a start-up stage, during which the system is operated in an initialization mode, before moving

into its operational mode. This start-up is modeled in Stateflow® [36] by the statechart [10] shown in Fig. 10.

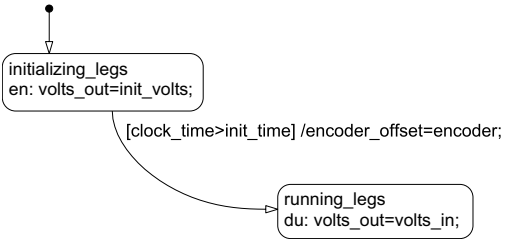


Fig. 10. Stewart platform start-up procedure

Once the required measurements have been taken, the Stewart platform must be returned to a safe and stable position. This is performed by the shut-down stage, modeled by the statechart in Fig. 11. After the operational mode, *running_legs*, a reset mode, *reset_legs*, is entered during which the extensions of the legs are reset. Next, the final mode, *done*, is entered during which all control signals are commanded to 0. Once this mode is entered, it is safe to turn off the system power.

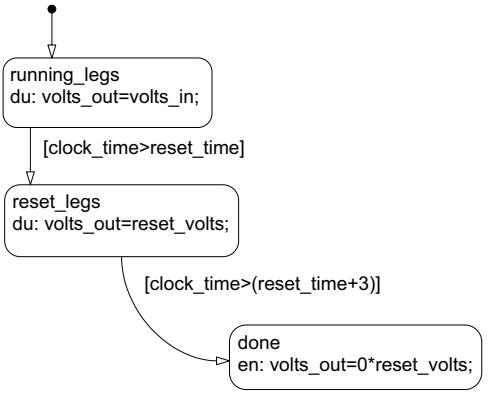


Fig. 11. Stewart platform shut-down procedure

6.5 Safety

Embedded control systems are fail-critical; they exhibit potentially dangerous behavior to the point where they may trigger catastrophic events. For this reason, the legs on the Stewart platform must not be driven beyond their maximum extension. This condition is ensured by a control that enforces a

hardware limit. The controller uses the current extension of each leg and the requested force to be exerted upon this leg as shown in the statechart in Fig. 12(a). If a leg is within 2% of its full extension, the controller will not allow further extension. The output of this statechart, *F_des_out*, is then passed through the force saturation computation statechart shown in Fig. 12(b).

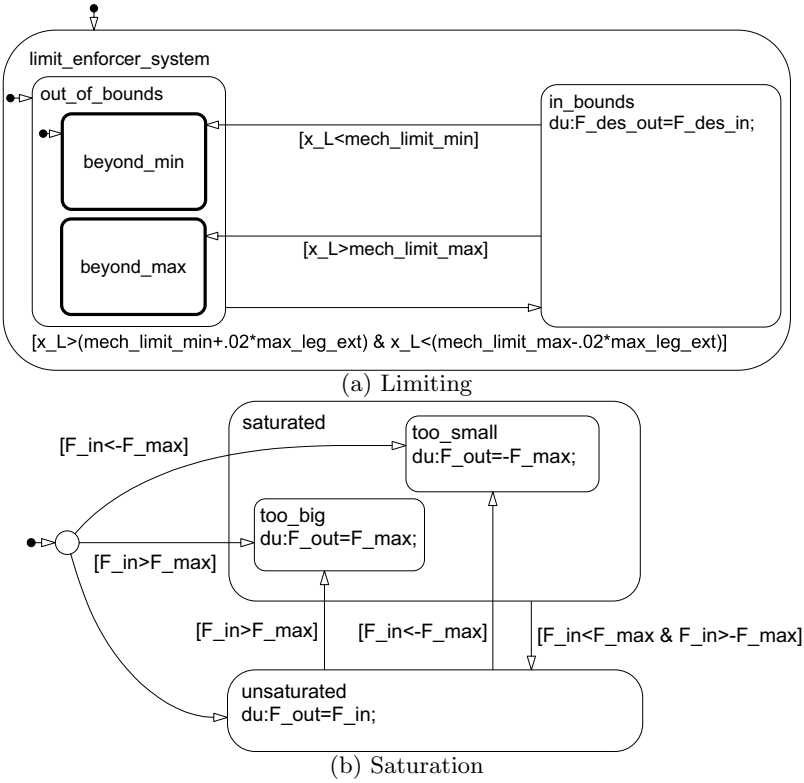


Fig. 12. Force safety computations

In many cases, emergency hardware is available that allows a safety switch to immediately invoke a safe controller. Often, this safe controller is nothing but a simple strategy to shut off power (the "big red button"). However, simply shutting off power is not always a feasible approach. For example, in aircraft, a hardware or software mode-switch may have to be present that has a proven controller (human or automatic) immediately take over. If the controller that is tested is designed to deal with calamities, it requires pushing the system to its envelope of safe behavior. In this case, the safety switch may even have to be made before the test of the prototype controller goes awry, as it is necessary to be in a recoverable state when the switch is made.

7 Plant Modeling

In many applications the equations of motion for the system can be described by first principles. Simulink and Stateflow provide the functionality to describe the overall architecture of a system. Tools for domain-specific physical behaviors include SimMechanics [30], for characterizing the dynamics of two- and three-dimensional mechanical components, and SimPowerSystems [11], for modeling the dynamics of electrical power systems.

The SimMechanics model of the overall Stewart platform mechanics was exported from a computer aided design (CAD) drawing in SolidWorks [35]. Fig. 13 shows a SimMechanics model of a Stewart platform leg, including friction. The gray area shows the SimMechanics part, which consists of a body representing the lower leg and a body representing the upper leg. The two bodies are connected by a prismatic joint. The *Lower Leg Sensor* and the *Upper Leg Sensor* sense the position and orientation of the lower and upper leg bodies for display with the Virtual Reality Toolbox [41]. The *Lower Connect* and *Upper Connect* ports connect the leg to the base and top plate of the Stewart platform, respectively. The prismatic joint is modeled to have stiction by the *Stiction Actuator*. This actuator takes forward and reverse stiction values and evaluates whether the static friction value is between these forward and reverse values. If it is, the joint is locked. Otherwise, the joint moves with kinetic friction and an external actuation force. The kinetic friction is computed by the *Friction* block and includes nonlinearities such as spring-damper endstop behavior.

The connections within the SimMechanics domain are energy connections that carry two conjugate variables, velocity and force. The product of these variables constitutes power [27]. The SimMechanics compiler automatically derives the computational direction of the velocity and force (the computational causality) that needs to be made explicit in a Simulink model [43]. Because the connections carry two variables that are computed from opposite ends, the connections are undirected, and, therefore, instead of an arrowhead, a direction-neutral line ending (such as a solid circle or a square) is used.

Typically, values for physical parameters such as moments of inertia, masses, rod lengths, and gear ratios are well known and can be incorporated into the model. Parameters such as friction coefficients, viscosity, and stiction behavior, however, are not precisely known. Measured data from the rapid prototyping system can be used with the first-principle description to calibrate these parameters using Simulink Parameter Estimation [6]. Simulink Parameter Estimation estimates parameters and dynamic states in Simulink and related modeling environments such as SimMechanics and SimPowerSystems. Simulink Parameter Estimation allows the selection of a set of parameters and states to be estimated. Minimum and maximum values of the parameters and initial states can be set, in addition to the expected values. Simulink Parameter Estimation uses optimization algorithms from the Optimization

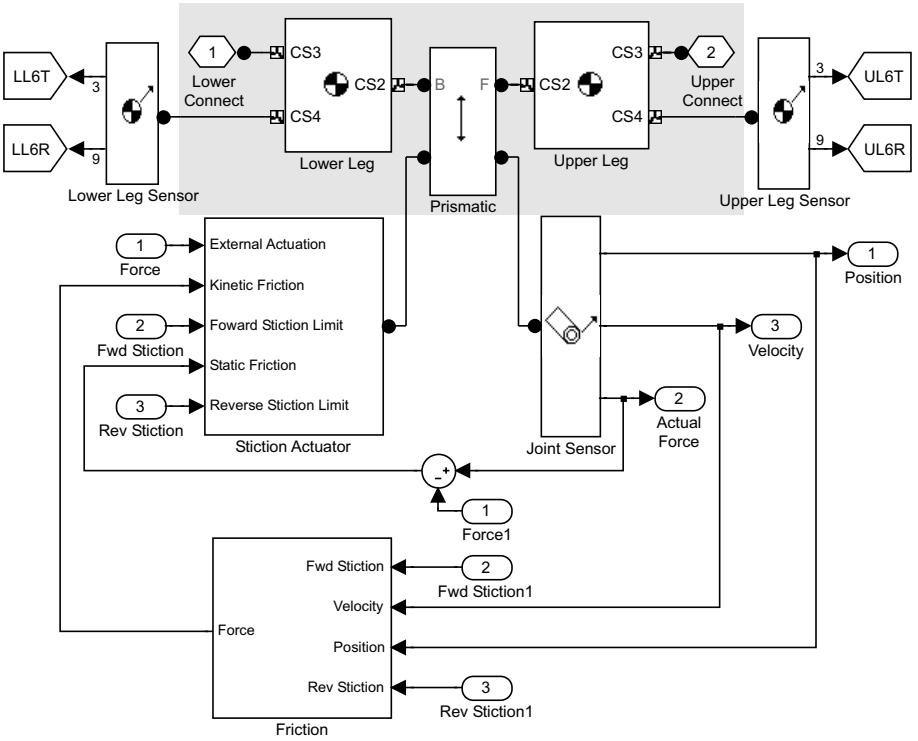


Fig. 13. A SimMechanics model of one leg of the Stewart platform

Toolbox [26] and, optionally, the Genetic Algorithm and Direct Search Toolbox [9].

Linear system identification tools are useful if the dynamics of a system behave in a near-linear manner about a given operating condition, but cannot be modeled from first principles. The System Identification Toolbox [38] computes linear discrete and continuous models using both time and frequency domain data.

Transfer function estimation is another process that takes experimental data and converts it using spectral estimation techniques to compute the frequency response of a system. The Signal Processing Toolbox [29] has many functions to support the estimation of a transfer function.

If linear models do not describe the model accurately and the underlying equations of motion are not well known, a nonlinear black box neural network can be created using the Neural Network Toolbox [21].

8 Data Acquisition

8.1 Experiment design

An experiment to acquire input/output data requires input signals that adequately excite the system. Experiment design must address [15, 25]: sample time selection, signal-to-noise ratio, and signal persistency.

Experiment validation and analysis is a post-processing step that involves the cleanup and initial analysis of the data. The post-processing steps include detrending, filtering, and outlier removal. Other experimental validation steps include [2, 6, 38]:

- Spectral estimation—the process of converting experimental data using spectral estimation techniques to compute the frequency response of a system. This type of estimation can be used to assess the order, the bandwidth, and a model for the system.
- Performing correlation calculations—to test for the existence of feedback, nonlinearities, and delays in the data.
- Data acquisition—two sets of data should be acquired from the rapid prototyping system: input for model estimation and validation. The form of the signals used in the validation data set should be different from the signals used for estimation data set generation.

8.2 Real-time needs in data acquisition

While data acquisition systems are useful for gathering data to build physical models, they cannot perform real-time data processing.

In a general data acquisition configuration, plant sensors provide a stream of measurements that are logged at a given sample rate. The plant may be excited by feedforward control, but the data acquisition system does not include any feedback control. In a real-time data acquisition configuration, closed-loop feedback control may pose stringent real-time constraints. The data acquisition part, however, can be run at a lower, and less demanding, frequency.

Rapid prototyping systems are preferable to standard data acquisition tools in the following cases:

- When the plant must be operated in a given mode to obtain data to estimate model parameters. For example, to determine the static friction of each of the legs of the Stewart platform, they have to be movable in both the positive and negative directions. An initial start-up phase is, therefore, necessary to extend all the legs to a point where they can move in both directions.
- When a plant is unstable and requires feedback to stabilize the dynamics for an experiment.

- When other control loops must be in operation while performing the experiment (e.g., when designing the spark control for a combustion engine while keeping the rest of the engine control strategy and I/O interfacing the same).
- When safety control functionality must be in place to protect the system.

8.3 Data acquisition techniques

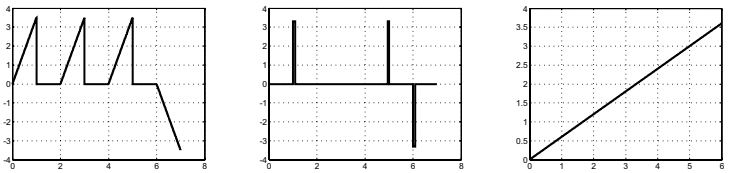
There are three basic techniques for data acquisition. Each has advantages and drawbacks [44]:

- *Polling* reads the status of a device regularly and is easiest to understand and debug.
- *Interrupt-based* data acquisition directs attention to a device only when it requests attention and is more flexible but suffers from "interrupt latency."
- *Direct memory access* (DMA) moves blocks of data directly into memory but requires the data to be processed as it comes in.

8.4 Parameter estimation

Once the acquired data has been uploaded from the target to the host (see Section 4), Simulink Parameter Estimation can be used to estimate the parameters of a plant model.

For example, in Fig. 14, three sets of data were acquired from the Stewart platform. The input is the voltage with which one of the piezoceramic actuators is driven. The output is the extension of the corresponding leg as measured by the grated slide.



(a) Data set 1 (b) Data set 2 (c) Data set 3

Fig. 14. Excitation

These data sets are used to estimate the static and dynamic friction behavior of one of the legs. Fig. 15 shows the response of the Stewart platform to each of the excitations in Fig. 14.

Fig. 16 shows the resulting output for the excitation in Fig. 14(a). The solid curve is the measured output, also shown in Fig. 15(a). The initial model output using parameter values as chosen by an educated guess is depicted by the dotted curve. The parameter values that result from the estimation

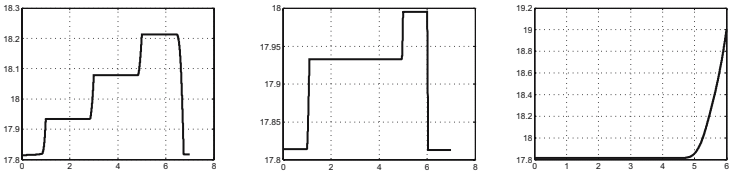


Fig. 15. Response

are given in Table 1. The corresponding trace is the dashed curve shown in Fig. 16. The estimated parameter values produce a model output that, while it better approximates the measured output, still deviates significantly from the actual output. Investigation of the model and the deviations reveals position-dependent stiction. This dependency can be added to the model structure and the parameter estimation process can be repeated.

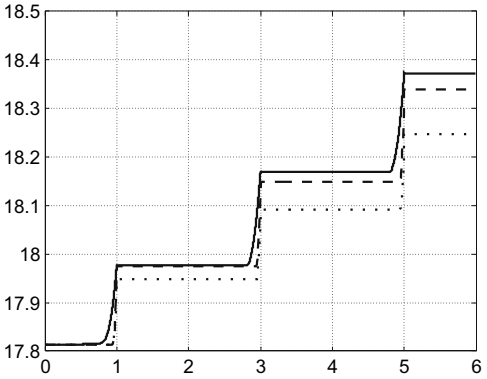


Fig. 16. Parameter estimation result

9 Stages in Control System Design

Once a model is available, the design of a control system can commence as follows:

1. Transform the model of the physical system into a form suitable for controller design. If the model is nonlinear, use the linearization tools to extract a linear model of the plant at various operating points [7, 32]. Simulink Control Design [32] provides tools to automatically extract a linear model from a Simulink block diagram.

Table 1. Estimated parameter set

Name	Value	Estimate	Initial Guess	Min.	Max.	Typical Value
Kf	0 .74713	√	Kf	0	+Inf	0 .75
Kfv	15	√	Kfv	0	+Inf	15
Kk	1 .0015	√	Kk	0	+Inf	1
Kv	40	√	Kv	0	+Inf	40
sf_f	-3 .5004	√	sf_f	-Inf	0	-3 .5
sf_r	3 .5	√	sf_r	0	+Inf	3 .5
sg_f	0 .99998	√	sg_f	0	+Inf	1
sg_r	1	√	sg_r	0	+Inf	1
xL_offset	17 .814		xL_offset	17 .814	+Inf	17 .814

2. Using a linear plant model, employ classical, modern, and robust control design tools to get a close estimate of the feedback control system components [7,17,24,34,46]. Linearization of a nonlinear model will usually result in a number of *modes of operation* for which different linearized models are derived. Such systems are called *hybrid dynamic systems*, and they require dedicated synthesis and analysis techniques [16, 40]. Care should be taken to account for computational delays and sampling effects. These variables can affect the stability and robustness of a control system design [37].
3. Implement the feedback control system design in the nonlinear Simulink plant model. Optimize controller performance on the full nonlinear model [13]. Simulink Response Optimization [33] lets users specify constraints on the response of the control system and pick parameters to optimize.
4. Simulate the nonlinear control and plant model to validate the design. A set of test cases that "lock down" required behavior may be used.
5. Generate code for the designed control and test this code against a nonlinear plant model or a real-time plant model. The real-time version tends to be more accurate in time but less accurate in terms of variable values because real-time simulation typically requires less accurate models to satisfy the response time constraints.
6. Test and tune the control system performance. Simulink can communicate directly with embedded software running on a target. Changes in parameters in the Simulink model from which the target embedded software is generated are communicated to the target application to take effect while the software is running.

10 The Stewart Platform Controller

A feedback control law was designed that drives the Stewart platform to a commanded position. In this feedback control law, the offset and hysteresis

need to be accounted for. If a proportional-integral-derivative (PID) controller were used, the proportional gain would be excessively high.

For the Stewart platform in Fig. 2, the offset at which the piezoceramic motors actually start moving is determined from acquired data. Because of the physics of the motors, a significant force is required to ensure that the leg starts moving. This force corresponds to approximately ± 3.2 V of command voltage, depending on the extension of the leg and the direction in which it is required to start moving. The velocity against voltage profile is linear once this offset is established, causing some hysteresis around 0 V.

The control algorithm used is shown in Fig. 17. The input and output of this control algorithm are six-dimensional variables, corresponding to the six legs of the hexapod. The algorithm takes as input the desired extension of each of the legs, x_Ldes, the error between the actual and desired extension of each of the legs, x_Le, and the time derivative of the actual extension, $xdot\_L$. The output of the algorithm is the desired force to be exerted by each motor that drives the extension of each leg, F_des. The desired force is the sum of a PID control that uses a first-order filter to approximate the derivative, an offset term to compensate for the stiction nonlinearity of the prismatic joints, and the derivative of the desired leg extension to improve tracking of the prescribed movement. Note that, to drive the motor, the computed force is translated into a voltage. This translation is not shown in Fig. 17.

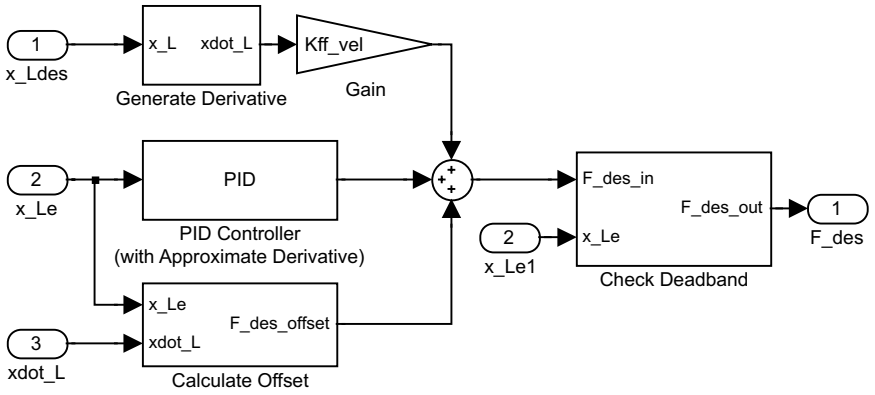


Fig. 17. Stewart platform feedback control

Using Simulink Response Optimization, the parameters for the respective control terms could be quickly estimated. If the performance of the implemented system does not meet the requirements, additional data can be acquired from the Stewart platform and used to refine the plant and controller models.

11 Conclusions

This chapter has discussed the use of MathWorks products for embedded control system design, with emphasis on rapid prototyping. xPC Target and xPC TargetBox play a central role in this process, as the use of general-purpose PC-based hardware makes them particularly well suited for prototyping applications. In addition, xPC Target can be equally well employed in other system configurations [19], providing hardware-in-the-loop simulation and testing capabilities to assist in the development of embedded controllers.

The characteristics of embedded control systems and the relevant features of xPC Target were discussed. The industrial PC, xPC TargetBox, was described in the context of a Stewart platform application.

Acknowledgements

The authors thank Mike Dickens, Tony Lennon, and Rosemary Oxenford for helpful comments on an earlier draft.

References

1. Karl J. Åström and Björn Wittenmark. *Computer Controlled Systems: Theory and Design*. Prentice-Hall, Englewood Cliffs, NJ, 1984.
2. Julius S. Bendat and Allan G. Piersol. *Random Data: Analysis & Measurement Procedures*. Wiley-InterScience, Hoboken, NJ, 2000.
3. CAN specification. Technical Report, 1991. Robert Bosch GmbH, Stuttgart, Germany.
4. Richard C. Dorf. *Modern Control Systems*. Addison-Wesley Publishing Co., Reading, MA, 1987.
5. Tom Erkkinen. How to use on-target rapid prototyping. `http://www.embedded.com/showArticle.jhtml;jsessionid=A3Q5VJ4OCNR3GQSNDBCCKH0CJUMEKJVN?articleID=51201234`, October 2004.
6. Simulink Parameter Estimation. *Simulink Parameter Estimation User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
7. Gene F. Franklin, J. David Powell, and Abbas Emami-Naeini. *Feedback Control of Dynamic Systems*. Prentice-Hall, Englewood Cliffs, NJ, 2002.
8. Gene F. Franklin, J. David Powell, and Michael L. Workman. *Digital Control of Dynamic Systems*. Prentice-Hall, Englewood Cliffs, NJ, 3^{rd} edition, 1997.
9. Genetic Algorithm and Direct Search Toolbox. *Genetic Algorithm and Direct Search Toolbox User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
10. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
11. Hydro-Québec TransÉnergie Technologies. *SimPowerSystems User's Guide*. The MathWorks, Inc., Natick, MA, 2004.

12. Applied Dynamics International. Introducing a target-based approach to rapid prototyping ECUs. Technical report, Applied Dynamics International, February 1997.
13. Hans-Dieter Joos. A methodology for multi-objective design assessment and flight control synthesis tuning. *Aerospace Science and Technology*, 3(3):161–176, 1999.
14. D.C. Karnopp, D.L. Margolis, and R.C. Rosenberg. *Systems Dynamics: A Unified Approach*. John Wiley & Sons, New York, 2^{nd} edition, 1990.
15. Lennart Ljung. *System Identification: Theory for the User*. Prentice-Hall, Englewood Cliffs, NJ, 2^{nd} edition, 1998.
16. Nancy Lynch and Bruce Krogh, editors. *Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*. Springer-Verlag, March 2000.
17. J.M. MacIejowski. *Multivariable Feedback Design*. Addison-Wesley Publishing Company, Reading, MA, 1989. Electronic Systems Engineering Series.
18. MATLAB. *The Language of Technical Computing*. The MathWorks, Inc., Natick, MA, 2004.
19. Pieter J. Mosterman, Sameer Prabhu, and Tom Erkkinen. An industrial embedded control system design process. In *Proceedings of the Inaugural CDEN Design Conference*, CD-ROM, Montreal, July 2004.
20. Pieter J. Mosterman, Janos Sztipanovits, and Sebastian Engell. Computer automated multiparadigm modeling in control systems technology. *IEEE Transactions on Control System Technology*, 12(2), March 2004.
21. Neural Network Toolbox. *Neural Network Toolbox User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
22. Gregory G. Nordstrom. Metamodeling —Rapid design and evolution of domain-specific modeling Environments. Ph.D. dissertation, Vanderbilt University, Electrical Engineering, Nashville, TN, May 1999.
23. Katsuhiko Ogata. *Discrete-Time Control Systems*. Pearson Education, Essex, United Kingdom, 2^{nd} edition, 1994.
24. Katsuhiko Ogata. *Modern Control Engineering*. Prentice-Hall Inc., Englewood Cliffs, NJ, 4^{th} edition, 2001.
25. Alan V. Oppenheim, Ronald W. Schafer, and John R. Buck. *Discrete-Time Signal Processing*. Prentice-Hall, Englewood Cliffs, NJ, 2^{nd} edition, 1999.
26. Optimization Toolbox. *Optimization Toolbox User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
27. Henry M. Paynter. *Analysis and Design of Engineering Systems*. The M.I.T. Press, Cambridge, MA, 1961.
28. Real-Time Workshop. *Real-Time Workshop User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
29. Signal Processing Toolbox. *Signal Processing Toolbox User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
30. SimMechanics. *SimMechanics User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
31. Simulink. *Using Simulink*. The MathWorks, Inc., Natick, MA, 2004.
32. Simulink Control Design. *Simulink Control Design User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
33. Simulink Response Optimization. *Simulink Response Optimization User's Guide*. The MathWorks, Inc., Natick, MA, 2004.

34. Sigurd Skogestad and Ian Postlethwaite. *Multivariable Feedback Control: Analysis and Design*. John Wiley & Sons, Inc., New York, 1996.
35. SolidWorks. *Introducing SolidWorks*. SolidWorks Corporation, Concord, MA, 2002.
36. Stateflow. *Stateflow User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
37. Brian Steven and Frank Lewis. *Aircraft Control and Simulation*. Wiley Interscience, New York, 2003.
38. System Identification Toolbox. *System Identification Toolbox User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
39. Agilent Technologies. *Agilent 54621A/22A/24A/41A/42A Oscilloscopes and Agilent 54621D/22D/41D/42D Mixed-Signal Oscilloscopes User's Guide*. Agilent Technologies Inc., Colorado Springs, CO, 2002. Publication Number 54622-97036.
40. Frits W. Vaandrager and Jan H. van Schuppen, editors. *Hybrid Systems: Computation and Control*, volume 1569 of *Lecture Notes in Computer Science*. Springer-Verlag, March 1999.
41. Virtual Reality Toolbox. *Virtual Reality Toolbox User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
42. K.C.J. Wijbrans. Twente hierarchical embedded systems implementation by simulation: a structured method for controller realization. Ph.D. dissertation, University of Twente, Enschede, The Netherlands, 1993. ISBN 90-9005933-4.
43. Giles D. Wood and Dallas C. Kennedy. Simulating mechanical systems in Simulink with SimMechanics. Technical Report 91124v00, The MathWorks, Inc., Natick, MA, 2003.
44. xPC Target. *xPC Target User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
45. xPC TargetBox. *xPC TargetBox User's Guide*. The MathWorks, Inc., Natick, MA, 2004.
46. Kemin Zhou and John C. Doyle. *Essentials of Robust Control*. Prentice-Hall Inc., Englewood Cliffs, NJ, 1997.

LabVIEW Real-Time for Networked/Embedded Control

John Limroth, Jeanne Sullivan Falcon, Dafna Leonard, and Jenifer Loy

National Instruments
11500 N. Mopac Expressway
Austin, TX 78759, U.S.A.
{john.limroth,jeannie.falcon,dafna.leonard,jenifer.loy}@ni.com

1 Introduction: Control Applications Using LabVIEW

Control applications are a very broad class of applications in which digital computers are used together with sensors and actuators to produce a desired behavior within the controlled system or process. This broad space of control applications can be roughly divided into two categories: industrial control and embedded control. Industrial or process control applications are those in which control is used as part of the process of creating or producing an end product. The control system is not a part of the actual end product itself. Examples include the manufacture of pharmaceuticals and the refining of oil. In the case of industrial process control, the control system must be robust and reliable, since the processes typically run continuously for days, weeks or even years.

Embedded control applications are those in which the control system is a component of the end product itself. For example, Electronic Control Units (ECUs) are found in a wide variety of products including automobiles, airplanes and home appliances. While embedded control systems must also be reliable, cost is a more significant factor, since the components of the control system contribute to the overall cost of manufacture of the product. In this case, much more time and effort is usually spent in the design phase of the control system to ensure reliable performance without requiring any unnecessary excess of processing power, memory, sensors, actuators, etc. in the digital control system.

LabVIEW has been used successfully for many years in test and measurement applications due to the ease of graphical programming and the wide array of functionality for interfacing directly to instruments, sensors and actuators. This connectivity to I/O has also enabled LabVIEW to be used for control applications. Because Windows is a non-deterministic operating system, LabVIEW for Windows was used to control relatively slow processes that

did not require fast update rates—for example, temperature control systems. For more information on LabVIEW applications, see [3].

With LabVIEW Real-Time, the power of LabVIEW has been extended by running LabVIEW applications under a deterministic real-time operating system. This has enabled development of high-speed closed-loop control applications with the ease of use of graphical programming. LabVIEW Real-Time can be used for industrial control applications using rugged hardware such as PXI or CompactFieldPoint. In addition, LabVIEW Real-Time can be used in the development of embedded control systems by acting as a rapid control prototype or as a hardware-in-the-loop (HIL) test system for testing embedded controller designs. The aim of this chapter is to explain how LabVIEW Real-Time and National Instruments (NI) real-time hardware targets can be used for the deployment of industrial control applications and the development of embedded control applications.

1.1 Industrial control: The programmable automation controller (PAC)

For the last three decades, with a proven track record for reliable operation, the programmable logic controller (PLC) has reigned supreme as the standard for automation and control applications. However companies turning toward flexible automation today are finding that PLCs are often not flexible enough for their rapidly changing production demands. In a typical plant there will be discrete control, motion control, visual inspection, process control, and production reporting. Traditionally, with a fixed automation model, each of these areas is viewed as an isolated discipline, is run on a separate controller, and is programmed with separate software. While this paradigm is workable in a fixed automation environment like Henry Ford's famous Model T line, where "Any customer can have a car painted any color that he wants so long as it is black," it is unsustainable in an environment where product customization and rapid changes in product design are the standards.

Faced with the need for the rapid evolution to high value production, in the last decade industry pundits and journal editors have foretold the widespread adoption of PC-based control. With features like powerful multidisciplinary software tools, floating-point processors, extensive memory, and a graphical interface, the PC seemed poised to become the ultimate industrial automation platform. However, today in most factories you'll see primarily PLCs performing machine and discrete control. To be fair, while many engineers have looked to the PC when they are incorporating advanced functionality like analog control and simulation, database connectivity, web-based functionality, and communication with third party devices, the PLC is still king for control.

The problem with PC-based control up to now is that standard PCs are not designed for rugged environments. Although some engineers use special industrial computers with hardened hardware and special operating systems,

most engineers avoid PCs for control because they, or someone they know, have had a reliability problem with a PC. PCs running standard operating systems with off-the-shelf hardware are generally too fragile and temperamental to be relied upon for embedded industrial control. In addition, the devices used within a PC for different automation tasks such as analog or digital I/O or motion may have different development environments.

Thus, many engineers either live without functionality, which cannot easily be accomplished with a PLC, or they cobble together a system that includes a PLC for the control portion of the code and a PC for the more advanced functionality. This is why, in many factories today, you will see PLCs being used in conjunction with PCs for data logging, connecting to bar code scanners, inserting information into databases, and publishing data to the web. The big problem with this type of setup is that these systems are often difficult to construct, troubleshoot, and maintain. The system engineer is left with the unenviable task of incorporating hardware and software from multiple vendors, and these components have not been designed to work together.

While experts were debating the advantages of PC-based control and PLC-based control, some vendors were designing new products that incorporate the best of both worlds. If the dot-com bubble had one good outcome it was a dramatic increase in processor speed, network reliability, and communication technology. Vendors today are incorporating industrial versions of floating-point processors, dynamic random access memory (DRAM), non-spinning memory storage such as CompactFlash, and fast Ethernet chipsets into industrial control products. More importantly, they are developing software with the flexibility and usability of PC-based control systems that can run under real-time operating systems for reliability.

Software in PC-based control systems today can perform multiple control disciplines, discrete control, process control, motion control, visual inspection, reporting, database connectivity, and statistics. This enables engineers to learn one software development tool, such as LabVIEW, and use it across the plant. The same software also has the ability to run on various platforms, from large mainframe installations to those embedded on a chip (Fig. 1). For instance, LabVIEW graphical programming software can run on Linux-based workstations, on Windows-based PCs, on embedded controllers running a real-time operating system (LabVIEW Real-Time), on personal digital assistants (PDAs) (LabVIEW PDA), and even compiled into silicon on a field-programmable gate array (FPGA) chip (LabVIEW FPGA). The software flexibility frees engineers from the constraints of environmental and reliability requirements by allowing them to select the most appropriate platform for their installation.

Programmable automation controllers

In December of 2002, the ARC Advisory Group identified this emerging class of software and hardware products and gave them the name “programmable

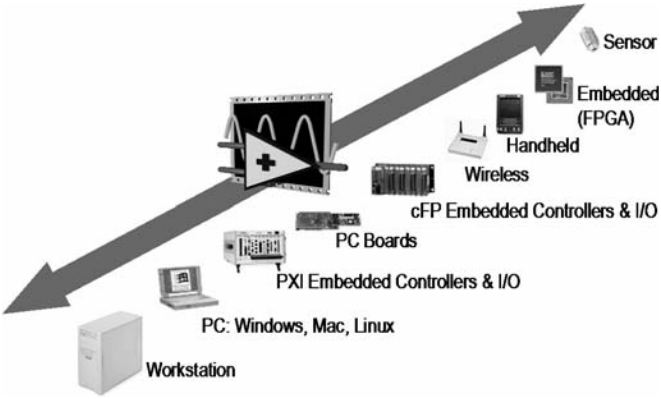


Fig. 1. LabVIEW can be used on a wide range of platforms and targets (workstation down to embedded devices)

automation controllers" or PACs [8]. They identified PACs as devices that offer open industry standards, extended domain functionality, a common development platform, and advanced capabilities.

LabVIEW Real-Time as the foundation for PAC

One example of a PAC is the combination of National Instruments LabVIEW Real-Time software and any of several National Instruments real-time hardware platforms. Together, these products deliver an embedded industrial control platform designed with PC flexibility and PLC reliability. These hardware platforms use LabVIEW Real-Time, which is LabVIEW running on a real-time operating system (RTOS). LabVIEW Real-Time extends the LabVIEW development environment to deliver deterministic, real-time performance. The engineer develops his application on a host computer using graphical programming and then downloads the application to run on an independent hardware target based on off-the-shelf computing components and an RTOS. The advantages of this approach are that the engineer can

- Develop reliable applications with graphical programming
- Implement and visualize precise deterministic performance
- Eliminate time spent integrating diverse I/O.

In the future, engineers will be able to use the same programming engine to distribute intelligence for automation to additional levels of control and devices. An example of this that exists today is the NI Compact Vision System (CVS). This device includes an x86-based processor which can run LabVIEW Real-Time. In addition, engineers can now re-program the digital I/O on the device using the LabVIEW FPGA.

PAC concepts can be extended by allowing for intelligence to be placed in new devices. Fig. 2 shows the extension of LabVIEW down to the level of FPGA chips. This technology is based on LabVIEW FPGA, which converts LabVIEW code into the VHDL code required to program these devices. LabVIEW FPGA can be used to delegate extremely time-critical functions to hardware such as limit and proximity sensor detection, safety interlocks, and sensor health monitoring. These types of functions require a very high degree of reliability which can only be achieved through implementation in silicon. In addition, custom digital protocols may be developed and implemented as well as simple, low-level control algorithms which must run at very high loop rates. LabVIEW FPGA allows a controls engineer to easily design their own hardware for these purposes.

1.2 Embedded control applications

The design of embedded control systems is often characterized using the "V-diagram" (see Fig. 2) The left side of this diagram shows the process beginning with requirements, through the design and simulation of the system to deployment of control software on an embedded target. The right side of the "V" shows the corresponding testing steps that are necessary as the functionality of the controller is verified and eventually incorporated into the overall designed system. LabVIEW and LabVIEW Real-Time can be used throughout this process to aid in the development of embedded control applications.

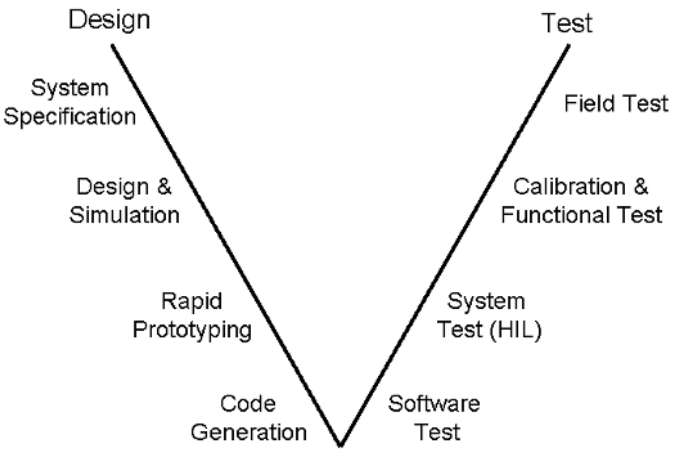


Fig. 2. V-diagram representing the embedded control design process

Design of embedded control systems

LabVIEW has a suite of powerful control design and simulation add-on modules for use in the design of embedded control systems. The native measurement functionality of LabVIEW can be used together with the system identification algorithms to characterize plant models directly from stimulus and response waveform data. Control design functions can be used to analyze plant models and systems and design control algorithms. The powerful user interface capability of the LabVIEW front panel can be used to create system identification and/or control design applications for a specific problem or class of problems that enables a repeatable, documented design process.

The LabVIEW Simulation Module provides a means of representing dataflow logic in the control block diagram form typical in the design of control systems, and it includes numerical ordinary differential equation solvers for simulation or real-time implementation. Plant models and control logic can be simulated off-line with complex inputs to refine the design before implementation on actual hardware. Models developed with LabVIEW can then be reused together with I/O functionality on Real-Time hardware for implementation or hardware-in-the-loop simulation.

Rapid control prototyping and hardware-in-the-loop (HIL) simulation

The increasing complexity of electronic control systems and the demands for faster time to market have led to an increase in the use of rapid prototyping and HIL testing in the design of embedded control systems. The complexity of these systems increases with the increasing functional demands required of applications such as that of modern automobiles. This in turn creates an increased need to prototype these systems before deployment to embedded controllers and to test controllers under simulated conditions before implementation on the actual physical plant system.

Faster time to market means that iterations of design cycles must be minimized while still ensuring that the designed system meets all requirements. Applying rapid control prototyping ensures that control designs are functionally correct before committing to the costly process of embedding the designed control algorithms in embedded control systems. By iterating new designs quickly using prototypes, the need for expensive redesigns after deployment to embedded controllers is minimized.

As described above, LabVIEW Real-Time and Real-Time hardware targets can be used to run models developed in LabVIEW for prototyping or HIL test applications. In addition, models or control logic designed with other software tools such as The MathWorks' Simulink[1] can be integrated with LabVIEW using the LabVIEW Simulation Interface Toolkit. LabVIEW's front panels

[1]Simulink® is a registered trademark of The MathWorks Inc.

can be used as a dynamic graphical user interface for Simulink applications during off-line simulation. The Simulink models can then be compiled into dynamic linked library (DLL) form using The MathWorks' Real-Time Workshop[2] together with the LabVIEW Simulation Interface Toolkit. The toolkit provides functions in LabVIEW that directly interface to the DLL generated from Real-Time Workshop. When combined with the I/O capability of LabVIEW, the compiled Simulink models can be run in LabVIEW Real-Time for implementation, prototyping, or HIL simulation.

1.3 Introduction to LabVIEW Real-Time

Most LabVIEW applications run on a general-purpose operating system (OS) like Windows, Linux, Solaris, or Mac OS. Some applications require deterministic real-time performance that a general-purpose OS cannot guarantee. The LabVIEW Real-Time Module extends the capabilities of LabVIEW to address the need for deterministic real-time performance. For more information on general programming with LabVIEW, see [2], [4].

The Real-Time Module combines LabVIEW graphical programming with the power of an RTOS, enabling the creation of deterministic real-time applications. Programs, referred to as Virtual Instruments (VIs), are developed in LabVIEW and executed on a real-time (RT) target. The RT target runs VIs without a user interface and offers a stable platform for real-time VIs.

Real-time basics

RTOSs are designed for high-reliability, deterministic systems. These operating systems differ from desktop operating systems in the following three ways:

1. OS scheduling mechanism ensures high-priority tasks always execute first
2. Software developer has explicit control over all system tasks
3. System does not require user input from peripherals such as mouse and keyboard.

In contrast, desktop OSs are designed to host a diverse set of applications including accounting software, desktop publishing, video gaming, and engineering tools. In addition, a desktop OS is expected to respond to all user inputs from the mouse and keyboard instantaneously. As a result, a desktop OS cannot be optimized for deterministic performance.

All LabVIEW Real-Time targets include an embedded RTOS that adheres to preemptive and round-robin scheduling, optimized for deterministic performance. With LabVIEW Real-Time, higher priority threads always preempt execution of lower priority threads. When threads of equal priority need to execute, round-robin scheduling gives each thread an equal amount of time

[2]Real-Time Workshop® is a registered trademark of The MathWorks, Inc.

with the processor. After one thread uses its available time slice, the OS automatically kicks that thread off the processor and the next thread in line begins executing. The combination of preemptive and round-robin scheduling ensures that LabVIEW Real-Time applications achieve deterministic performance with minimal jitter.

LabVIEW Real-Time Module platforms

National Instruments designed the LabVIEW Real-Time Module to execute VIs on two different real-time platforms. The LabVIEW Real-Time Module can execute VIs on hardware targets running the RTOS of the Venturcom Phar Lap Embedded Tool Suite (ETS) and on computers running the Venturcom Real-Time Extension (RTX).

Venturcom Phar Lap ETS provides an RTOS that runs on NI RT Series hardware to meet the requirements of embedded applications that need to behave deterministically or have extended reliability requirements.

Venturcom RTX adds a real-time subsystem (RTSS) to Windows. Venturcom RTX enables Windows and the RTSS to be run at the same time on the same computer. The RTSS has a priority-based real-time execution system independent of the Windows scheduler. RTSS scheduling supersedes Windows scheduling to ensure deterministic real-time performance of applications running in the RTSS.

Real-Time system components

A real-time system consists of software and hardware components. The software components include LabVIEW, the RT Engine, and VIs built using LabVIEW. The hardware components of a real-time system include a host computer and an RT target. The following sections describe the different components of a real-time system.

Host computer
The host computer is a computer with LabVIEW and the LabVIEW Real-Time Module installed on which VIs are developed for the real-time system. After the real-time system VIs are developed, they may be downloaded and run on the RT target. The host computer can run VIs that communicate with the VIs running on the RT target.

LabVIEW
VIs are developed with LabVIEW on the host computer. The Real-Time Module extends the capabilities of LabVIEW to allow selection of an RT target on which to run VIs.

RT Engine
The RT Engine is a version of LabVIEW that runs on the RT target. The RT Engine runs the VIs downloaded to RT targets. The RT Engine provides deterministic real-time performance for the following reasons:

- The RT Engine runs on an RTOS or RTX subsystem, which ensures that the LabVIEW execution system and other services adhere to real-time operation.
- The RT Engine runs on RT Series hardware or the RT target on the RTX subsystem. Other applications or device drivers commonly found on the host computer do not run on RT targets. The absence of additional applications or devices means that a third-party application or driver does not impede the execution of VIs.
- RT targets on which the RT Engine runs do not use virtual memory, which eliminates a major source of unpredictability in deterministic systems.

RT target
An RT target refers to RT Series hardware or the RTSS that runs the RT Engine and VIs created using LabVIEW. There are three types of RT targets: RT Series plug-in devices, networked RT Series devices, and the RTSS.

2 LabVIEW Real-Time Hardware Platforms

2.1 Real-Time hardware architecture

All LabVIEW Real-Time deployment platforms are based on a common hardware and software architecture. Each hardware target uses off-the-shelf computing components such as a microprocessor, RAM, non-volatile storage, and an I/O bus interface. The embedded software consists of an RTOS, driver software, and a specialized version of the LabVIEW run-time engine. See Fig. 3.

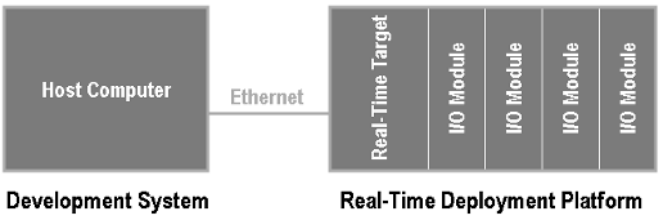


Fig. 3. LabVIEW Real-Time hardware architecture

While the core architecture is the same across all LabVIEW Real-Time targets, the extent to which these benefits can be realized varies according to the platform selected. Peripheral component interconnect (PCI) and PCI eXtensions for instrumentation (PXI) systems deliver the highest performance with minimal jitter while Compact FieldPoint and Compact Vision Systems deliver the highest degree of ruggedness. All platforms function equally as autonomous or stand-alone, headless systems.

PCI real-time systems

Many test and measurement applications today are based on PCI systems. LabVIEW Real-Time can be used with these systems by either adding a real-time component to a Windows system or converting a desktop PC to a dedicated real-time target.

Real-Time PCI plug-in boards

The NI PCI-7041/6040E RT Series plug-in board adds a real-time component to an existing test and measurement system such as a safety shutdown system integrated with a test application. The PCI-7041/6040E consists of two boards—an intelligent board and a multifunction data acquisition (DAQ) board—permanently joined together. The intelligent board plugs into a PCI slot in a Windows computer. It has the same basic components as a computer motherboard—an embedded microprocessor, RAM, and nonvolatile storage. The multifunction DAQ board includes connectivity for 16 analog inputs, two analog outputs, 8 digital input/output lines, and two counter/timer signals.

Using LabVIEW Real-Time, programs can be downloaded to the processor on the intelligent board, which runs an RTOS. The program runs on the embedded processor independent of all operations on the Windows processor. As long as the RT Series DAQ board has power, the embedded application will continue to run. As a result, real-time PCI plug-in boards are ideal for adding a reliable control component to a Windows-based system.

Desktop PCs

If more I/O is needed than is offered on the PCI-7041/6040E, a standard desktop PC can be converted to a real-time system using LabVIEW Real-Time for ETS Targets or LabVIEW Real-Time for RTX Targets.

Using LabVIEW Real-Time for ETS Targets, a certified desktop PC can be converted into a dedicated real-time hardware target. In this case, Venturcom Phar Lap ETS, a dedicated RTOS consisting of a single real-time kernel, is downloaded to the desktop PC microprocessor. Applications developed for a dedicated RTOS are developed on a separate host computer and then downloaded to the real-time target. This same architecture is used for all NI real-time hardware targets.

In some cases, a solution is needed in which a single machine provides a user interface in Windows as well as real-time control functionality. Using LabVIEW Real-Time for RTX Targets, a real-time component can be added to a variety of Windows desktop PCs. In this case, Venturcom RTX, an extension-based RTOS, is downloaded to the desktop PC microprocessor. An RTOS extension consists of a real-time kernel and a Windows kernel sharing the same processor. With this dual-kernel architecture, both the host application and real-time system can be run on the same machine.

With the RTX architecture, the real-time tasks are given higher priority; Windows tasks execute only when all real-time tasks are sleeping. However, the real-time and Windows applications still share the same hardware. If Windows initiates an operation that reserves the hardware for a set period of time

(for example, reserving the data bus for a large data transfer from the CD ROM), the real-time tasks will be unable to use that resource until the operation is complete. This type of situation can cause a priority inversion to occur between the real-time and Windows tasks. Therefore, applications running on the machine must be implemented to carefully avoid this type of resource contention, possibly limiting the functionality of applications running in the Windows environment.

PXI/CompactPCI real-time systems

In many applications a system is needed that is more rugged and provides more I/O capabilities than a standard desktop PC. Real-time PXI systems consist of a rugged chassis, embedded controller, and plug-in I/O modules. This target is ideal for high-performance systems such as HIL test of electronic control units and vibration monitoring for machine condition monitoring applications.

Using LabVIEW Real-Time for ETS Targets, the embedded controller can be converted to a real-time controller by downloading the RTOS and application software to a dedicated microprocessor. The embedded software then has access to all I/O in the PXI system, taking advantage of the PXI advanced timing and synchronization features to achieve precise I/O triggering and multimodule synchronization.

FieldPoint and Compact FieldPoint real-time systems

In industrial control applications, which are often highly distributed, real-time control is needed in a small, ruggedized form factor suitable for harsh environments. FieldPoint and Compact FieldPoint systems consist of a controller with an embedded processor running an RTOS and a variety of I/O modules. These systems feature rugged hardware, designed to operate in harsh industrial environments. In addition, the software structure of FieldPoint and Compact FieldPoint systems support a built-in publish/subscribe protocol, making these ideal for creating distributed applications.

Using LabVIEW Real-Time for ETS Targets, distributed measurement and control applications can be developed and then downloaded to run on the FieldPoint or Compact FieldPoint controller. The embedded software then has access to all I/O connected to the controller.

Compact Vision real-time systems

Compact Vision Systems are small rugged systems optimized for machine vision application such as automated inspection. A single small Compact Vision System includes an embedded processor with an RTOS, connections for three IEEE 1394 DCAM cameras, a local video display, an Ethernet port, 15 digital inputs, and 14 digital outputs.

Using LabVIEW Real-Time for ETS Targets, machine vision applications can be developed and then downloaded to run on the Compact Vision System. The embedded software then has access to the camera signals and digital I/O connected to the system. In addition, LabVIEW FPGA can be used to customize the operation of the digital I/O lines, converting the lines into advanced timing and triggering for complementary machinery or a custom communication protocol to transfer data to proprietary external equipment.

CompactRIO reconfigurable control and acquisition system

For control prototyping or industrial control applications that require a high-performance control system in a compact, ruggedized form factor, NI CompactRIO may be used. CompactRIO is an advanced reconfigurable embedded system development platform that offers flexibility, performance, and low-level access to reconfigurable hardware resources. Sophisticated embedded control or acquisition systems developed with CompactRIO match or exceed the performance and optimization of custom-designed hardware devices. CompactRIO employs a user-programmable FPGA core that automatically synthesizes an optimized custom hardware circuit implementation of a LabVIEW FPGA application to implement any input, output, communication, or control design.

The reconfigurable chassis is the heart of the CompactRIO system and contains the Reconfigurable I/O (RIO) FPGA core. This user-defined RIO FPGA is a custom hardware implementation of a control logic, input/output, timing, triggering, and synchronization design. The RIO FPGA circuit is connected to the I/O modules in a star topology, allowing direct access to each module for precise control and unlimited flexibility in timing, triggering, and synchronization. The RIO FPGA core has built-in data transfer mechanisms to pass data to a host processor for real-time analysis, post processing, data logging, or display in a real-time or Windows host application.

The NI CompactRIO Real-Time Controller has a powerful floating-point processor which runs embedded LabVIEW Real-Time applications for tightly integrated analog process control, batch control, motion, signal processing, data logging, and Ethernet or serial communication.

2.2 LabVIEW Real-Time deployment platform comparison

Each LabVIEW Real-Time deployment platform is designed for a slightly different application. PCI and PXI systems provide the highest performance while Compact FieldPoint and Compact Vision Systems deliver the most rugged hardware. In this section, we compare key aspects of each LabVIEW Real-Time hardware target in areas of I/O availability, performance, and physical attributes.

I/O availability

PXI/CompactPCI and Desktop PCs running ETS RTOS deliver the highest variety of I/O with NI and third-party modules providing signal connectivity to analog, digital, counter/timer, image, motion control, reflective memory, serial, general-purpose interface bus (GPIB), Controller Area Network (CAN), and more. While these systems provide the highest variety of I/O, the programming is more complicated than that for the FieldPoint and Compact FieldPoint systems. With FieldPoint and Compact FieldPoint, the I/O timing is defined by the I/O hardware; therefore, the application program interface (API) is less involved than what is required from the other platforms.

Desktop PCs running ETS or RTX RTOS, PXI, and Compact Vision Systems also work with LabVIEW FPGA targets, NI Reconfigurable I/O hardware programmed through LabVIEW software. With this unique level of customizability, engineers can define hardware operation for custom digital protocols, onboard processing, and immediate decision making.

All LabVIEW Real-Time systems are designed to work well in a distributed architecture, with multiple nodes connected via Ethernet. However, Desktop PCs with ETS RTOS and PXI systems deliver software-transparent I/O expandability with NI MXI. Using MXI, two or more PXI chassis can be daisy-chained together so that all I/O is controlled by a single PXI or Desktop PC controller.

Performance

Performance of LabVIEW Real-Time systems can be measured in terms of deterministic execution, I/O timing, triggering, synchronization, and processor speed. Determinism is the most fundamental component of all real-time systems. It is defined by how consistently a system is able to perform a given operation within a fixed amount of time. Determinism is affected by the operating system, software program architecture, and integration of application software with I/O timing and synchronization capabilities. The processor speed determines the minimum loop-cycle time.

PXI, PCI, and desktop PC real-time systems

NI PXI and PCI real-time platforms feature the fastest processors and a data bus optimized for high-speed data transfers and synchronized timing across multiple devices. All timing for these devices is configured programmatically. Using LabVIEW and the driver API, the sample frequency of input signals and the update rate of output signals are specified.

Using the PXI trigger bus, the benefits of hardware timing can be extended to synchronize operations across two or more I/O modules. PXI defines eight highly flexible bused trigger lines that can be used in a variety of ways. For example, triggers can be used to synchronize the operation of several PXI modules. In other applications, one module can control carefully

timed sequences of operations performed on other modules in the system. In addition, trigger signals can be shared between multiple modules for deterministic responses to asynchronous external events that are being monitored or controlled. The number of triggers that a particular application requires varies with the complexity and number of events involved.

System jitter can be further reduced using the PXI star trigger bus. The star trigger bus implements a dedicated trigger line between the Star Trigger Slot (first peripheral slot adjacent to the System Slot) and the other peripheral slots. PXI star trigger delivers two unique advantages in augmenting the bused trigger lines. The first is a guarantee of a unique trigger line for each module in the system. For large systems, this eliminates the need to combine several module functions on a single trigger line or to artificially limit the number of trigger times available. The second advantage is the low-skew connection from a single trigger point. The PXI backplane defines specific layout requirements so that the star trigger lines provide matched propagation time from the star trigger slot to each module for very precise trigger relationships between each module.

FieldPoint and Compact FieldPoint real-time systems
FieldPoint and Compact FieldPoint systems are designed to be small and compact. In addition, each I/O module has a unique fixed sampling rate determined by the hardware module. As a result, the I/O programming for these systems is considerably less involved than that for the PXI and PCI counterparts.

Because I/O timing for FieldPoint and Compact FieldPoint systems is fixed within each hardware module, these systems have three asynchronous loops executing in parallel:

- Input channel updates sent from the input module to the software control loop
- Software control loop running on the embedded controller
- Output channel updates sent from the software control loop to the output module.

As a result, these systems are best suited for applications requiring a control loops rate of less than 200 Hz.

Physical attributes

In addition to deterministic performance, the RTOSs impart a higher level of reliability because they are specialized, streamlined operating systems that use fewer resources and eliminate the fragilities of standard operation systems. Along with the more reliable software framework, LabVIEW Real-Time platforms include hardware modifications that deliver an even higher level of reliability suitable for industrial environments.

National Instruments offers a variety of rugged real-time hardware platforms that extend the application space for LabVIEW Real-Time into harsh environments where the hardware must withstand extreme temperatures, shock, vibration, and intermittent power failures. Compact FieldPoint, Compact Vision System, and the PXI-8145 RT embedded controller are designed with no moving parts, eliminating the most common failures due to shock and vibration.

Compact FieldPoint controllers are designed to work in a wide range of temperatures, from -25 to 60 ^{o}C, thus eliminating failures due to heat. They also feature redundant power supply inputs for a seamless connection to back-up battery. In addition, Compact FieldPoint systems have been certified to meet the following industrial standards:

- Low-Voltage Directive, European Directives for CE Marking (73/23/EEC)
- European Union (EN) and International (IEC) Safety Standards for Electrical Equipment for Test and Measurement, Control, or Laboratory (EN 61010-1, IEC 61010-1)
- Process Control Equipment (UL 3121-1, UL 61010C-1)
- Safety Requirements for Electrical Equipment for Measurement, Control and Laboratory Use (CAN/CSA C22.2 No. 1010.1)
- Hazardous Locations (Class I, Division 2, Zone 2).

Compact FieldPoint and Compact Vision Systems provide the best portability because of the small rugged design. Compact FieldPoint features a rigid backplane with screw fasteners for the I/O modules.

3 LabVIEW Real-Time Software Architecture

3.1 Architecture of a deterministic application in LabVIEW Real-Time

LabVIEW Real-Time relies on multithreading and a strict priority scheme to guarantee determinism to critical tasks.

To create a multithreaded application in LabVIEW, time-critical tasks must be separated from non-time-critical tasks. VIs can then be built to complete each task. The VIs are prioritized and then categorized into one of the available execution systems to control the amount of processor resources each VI receives. LabVIEW assigns each VI to an execution system thread according to the VI priority and execution system assigned.

The threads execute on the processor accordingly. Deterministic communication methods can be used to pass data between the different VIs.

The RTOS on RT targets and the RTSS use a combination of round-robin and preemptive scheduling to execute threads in the execution system. Round-robin scheduling applies to threads of equal priority. Equal shares of processor time are allocated among equal priority threads. For example, each

normal priority thread is allotted 10 ms to run. The processor executes all the tasks it can in 10 ms, and whatever is incomplete at the end of that period must wait to complete during the next allocation of time. Conversely, preemptive scheduling means that any higher priority thread that needs to execute immediately pauses execution of all lower priority threads and begins to execute. A time-critical priority thread is the highest priority and preempts all priorities. For more information on creating deterministic LabVIEW Real-Time applications, refer to [6] and [7].

3.2 Dividing tasks to create deterministic multithreaded applications

Deterministic control applications depend on time-critical tasks to complete on time, every time. Therefore, time-critical tasks need enough processor resources to ensure their completion. Time-critical tasks must be separated from all other tasks in the application and placed in a separate VI to ensure that they receive enough processor resources. For example, if a control application processes measurement data at regular intervals and stores the data on disk, the timing and control of the data acquisition must be handled in a time-critical VI. However, storing the data on disk is inherently a non-deterministic task because file I/O operations have unpredictable response times that depend on the hardware and the availability of the hardware resource. Therefore, file I/O operations must be placed in a normal priority VI.

The time-critical priority VI receives the processor resources necessary to complete the task and does not relinquish control of the processor until it cooperatively yields to the normal priority VI or until it completes the task. The normal priority VI then runs until preempted by the time-critical VI. Deterministic methods can be used to pass data between the VIs running on the RT target.

The following VI priorities, listed in order from lowest to highest, can be selected to assign VIs to an execution system thread:

- background priority (lowest)
- normal priority
- above normal priority
- high priority
- time-critical priority (highest)

Threads of higher priority preempt threads of lower priority. Normal priority is the default thread priority for all VIs created in LabVIEW. The time-critical priority preempts all thread priorities. A time-critical priority thread does not relinquish processor resources until it completes all tasks. However, a time-critical thread can explicitly relinquish control of processor resources to ensure that the thread does not monopolize the processor resources. SubVIs inherit the priority of the caller VI. For example, a subVI called in a time-critical VI runs in time-critical priority. Because time-critical priority threads

cannot preempt each other, only one time-critical thread should be created in an application to guarantee deterministic behavior.

Because of the preemptive nature of time-critical VIs, they can monopolize processor resources. A time-critical VI might use all of the processor resources, not allowing lower priority VIs in the application to execute. Time-critical VIs must periodically yield, or sleep, to allow lower priority tasks to execute without affecting the determinism of the time-critical code. By timing control loops, time-critical VIs can yield and cooperatively relinquish processor resources. LabVIEW Real-Time provides several ways to yield a thread; simple VIs can be used to yield periodically until the CPU time reaches a certain count. Loop timing can also be tied to hardware interrupts. Finally, the Timed Loop can be used to configure any of these options for optimal performance or if multi-rate applications are needed.

3.3 Passing data between VIs

Because tasks must be separated into time-critical VIs and non-time-critical VIs, data must be communicated between the separate VIs in a typical LabVIEW Real-Time application. LabVIEW provides many different ways of sharing data between loops, such as global variables, functional global variables, and the Real-Time FIFO VIs (FIFO stands for first-in-first-out) to send and receive data between VIs in an application. The Real-Time FIFO VIs are always safe to use in a time-critical VI, regardless of the size of the data.

The Real-Time FIFO VIs can be used to transfer data between VIs in an application. An RT FIFO acts like a fixed queue, where the first value written to the FIFO is the first value that can be read from the FIFO. RT FIFOs and LabVIEW queues both transfer data from one VI to another. However, unlike a LabVIEW queue, an RT FIFO ensures deterministic behavior by imposing a size restriction on the data. The number and size of the RT FIFO elements must be specified. Both a reader and writer can access the data in an RT FIFO at the same time, allowing RT FIFOs to work safely from within a time-critical VI.

Because of the fixed-size restriction, an RT FIFO can be a lossy communication method. Writing data to an RT FIFO when the FIFO is full overwrites the oldest element. Data stored in an RT FIFO must be read before the FIFO is full to ensure the transfer of every element without losing data.

3.4 Communicating with applications on an RT target

The RT Engine on the RT target does not provide a user interface for applications. One of two communication protocols can be used, front panel communication or network communication, to provide a user interface on the host computer for RT target VIs.

Front panel communication

A LabVIEW VI consists of two parts: a *front panel*, which serves as the user interface for the application, and a *block diagram*, which consists of the graphical code that defines program execution. With front panel communication, LabVIEW and the RT Engine execute different parts of the same VI, as shown in Fig. 4. LabVIEW on the host computer displays the front panel of the VI while the RT Engine executes the block diagram. A user interface thread handles the communication between LabVIEW and the RT Engine.

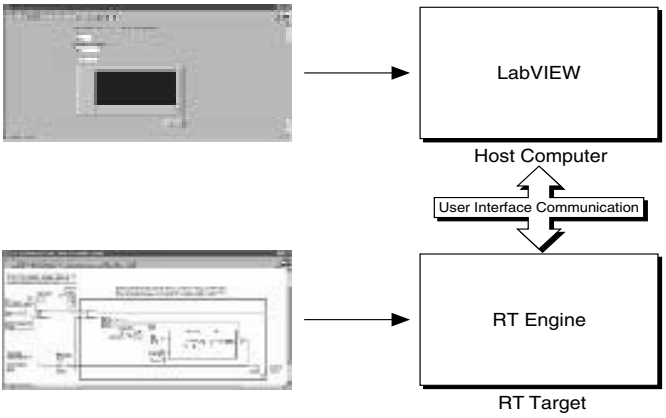


Fig. 4. Front panel communication protocol

Front panel communication can be used between LabVIEW on the host computer and the RT Engine to control and test VIs running on an RT target. After the VI has been downloaded and run, LabVIEW can be kept open on the host computer to display and interact with the front panel of the VI. Front panel communication can also be used to debug VIs while they run on the RT target. LabVIEW debugging tools—such as probes, execution highlighting, breakpoints, and single stepping—can be used to locate errors on the block diagram code.

Front panel communication is a good communication method to use during development because front panel communication is a quick method for monitoring and interfacing with VIs running on an RT target. However, front panel communication is not deterministic and can affect the determinism of a time-critical VI. Network communication methods should be used to increase the efficiency of the communication between a host computer and VIs running on the RT target.

Network communication

With network communication, a host VI runs on the host computer and communicates with the VI running on the RT target using specific network communication methods such as transmission control protocol (TCP), VI Server, and, in the case of non-networked RT Series plug-in devices, shared memory reads and writes. Network communication might be used for the following reasons:

- Another VI must be run on the host computer.
- The data exchanged between the host computer and the RT target must be controlled. Communication code can be customized to specify which front panel objects get updated and when. One can also control which components are visible on the front panel because some controls and indicators might be more important than others.
- Timing and sequencing of the data transfer must be controlled.
- Additional data processing or logging must be performed.

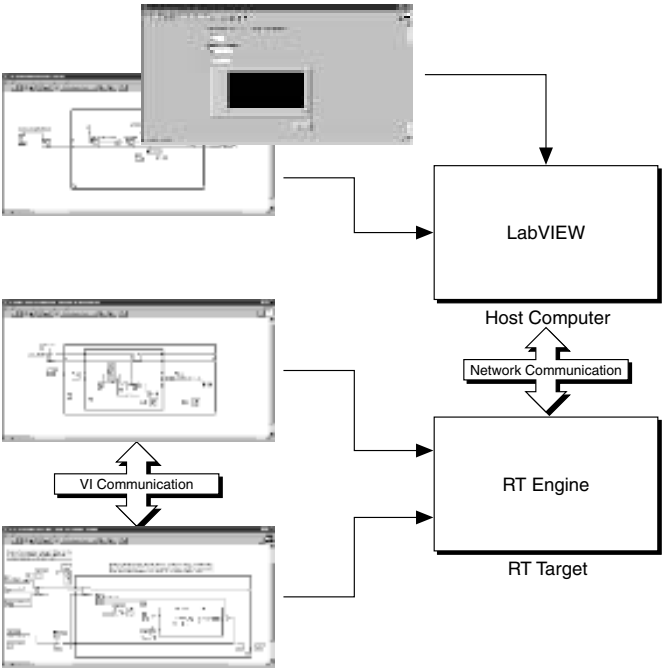


Fig. 5. Network communication protocol

In Fig. 5, the RT target VI is similar to the VI in Fig. 4 that runs on the RT target using front panel communication to update the front panel controls and

indicators. However, the RT target VI in Fig. 5 uses Real-Time FIFO VIs to pass data to a communication VI. The communication VI then communicates with the host computer VI using network communication methods to update controls and indicators.

Creating communication VIs with the RT Communication Wizard

The RT Communication Wizard can be used to greatly simplify the development of LabVIEW Real-Time applications that require network communication. The RT Communication Wizard creates VIs that deterministically transfer front panel control and indicator data from time-critical VIs running on an RT target to a VI running on the host computer. After a time-critical VI is specified, the RT Communication Wizard returns a list of controls and indicators present in the VI. The RT Communication Wizard replaces the front panel controls and indicators selected from the time-critical VI with Real-Time FIFO VIs. The RT Communication Wizard creates a normal priority VI that contains Real-Time FIFO VIs to send and receive front panel data from the time-critical VI. The Real-Time FIFO VIs transfer data deterministically and do not affect the timing of the time-critical VI. The RT Communication Wizard creates the following three VIs:

- Time-Critical VI—Runs on the RT target and contains the time-critical tasks and Real-Time FIFO VIs to transfer front panel data deterministically to the normal priority VI.
- Normal Priority VI—Runs on the RT target and contains all non-deterministic network communication tasks to update the host VI with front panel data received from the time-critical VI.
- Host VI—Runs on the host computer and displays the front panel controls and indicators of the time-critical VI.

Network and bus communication

High-level software protocols can be used to communicate between VIs running on the RT target and VIs running on a host computer. Each protocol has its advantages and disadvantages. The following list classifies the different communication methods:

- Shared memory communication—Used for communication between LabVIEW and RT Series plug-in devices or the RT target on the RTX subsystem.
- Network communication—Used for communication over Ethernet networks.
  - TCP
  - User Datagram Protocol (UDP)
  - DataSocket

  - VI Server
  - Simple Mail Transfer Protocol (SMTP) (send only)
- IrDA wireless communication—Used for communication with RT targets using Infrared Data Association hardware.
- Bus communication—Used for communication over different bus communication ports.
  - Serial
  - CAN
- Reflective Memory—Used for deterministic communication over a dedicated copper or optical network.

All of the methods described, with the exception of CAN and reflective memory, are non-deterministic and using them inside a time-critical VI adds jitter to the application. For additional information on networking in LabVIEW and LabVIEW Real-Time applications, see [1],[3],[5].

Shared memory
In operating systems like Windows, two processes or applications can communicate with each other using the shared memory mechanism of the operating system. Similarly, VIs running on an RT target and VIs running on the host computer can communicate using the Real-Time Shared Memory VIs. The Real-Time Shared Memory VIs can be used to read and write to shared memory locations of RT Series plug-in devices or the shared memory locations of the RTSS.

The Real-Time Shared Memory VIs communicate data deterministically because they have low overhead. However, the NI RT Series PCI-7041 plug-in devices have a shared memory size limit of 512 KB. If several megabytes of data are to be transferred, the data must be divided into smaller portions and then transferred. In doing so, the data in the shared memory must not be overwritten before it is read.

TCP
TCP is an industry-standard protocol for communicating over networks. VIs running on the host computer can communicate with RT target VIs using the LabVIEW TCP functions. However, TCP is non-deterministic, and using TCP communication inside a time-critical VI might cause the loop cycle time to vary from the desired time. The Real-Time Module extends the capabilities of the existing TCP functions to enable communication with networked RT Series devices and to allow communication across shared memory with RT Series plug-in devices.

UDP
UDP is a network transmission protocol for transferring data between two locations on a network. UDP is not a connection-based protocol, so the transmitting and receiving computers do not establish a network connection. Because there is no network connection, there is little overhead when transmitting

data. However, UDP is non-deterministic, and using UDP communication inside a time-critical VI might cause the loop cycle time to vary from the desired time. When using UDP to send data, the receiving computer must have a read port open before the transmitting computer sends the data.

Data can be transferred bidirectionally with UDP. With bidirectional data transfers, both computers specify a read and write port and transmit data back and forth using the specified ports. Bidirectional UDP data transfers can be used to send and receive data from the network communication VI on the RT target.

UDP has the ability to perform fast data transmissions deterministically. However, UDP cannot guarantee that all datagrams arrive at the receiving computer. Because UDP is not connection based, the arrival of datagrams cannot be verified. Network congestion must be avoided to ensure that the transmission of datagrams is not affected. Also, data stored in the data buffer of the receiving computer must be read fast enough to prevent overflow and loss of data.

DataSocket

DataSocket is an Internet programming technology to share live data between VIs and other computers. A DataSocket Server running on a host computer acts as a data repository. Data placed on the DataSocket Server becomes available for clients to access.

One advantage of using DataSocket is that multiple clients can access data on the DataSocket Server. A LabVIEW VI can use the DataSocket Write VI to post data to the DataSocket Server. Any number of VIs running on different RT targets or host computers can use the DataSocket Read VI to retrieve the data. RT target VIs can post data, such as status information, to the DataSocket Server for a VI running on a host computer to read.

DataSocket is non-deterministic and using DataSocket functions inside a time-critical VI adds jitter to the application.

VI Server

The VI Server can be used to monitor and control VIs on an RT target. Using VI Server technology, a LabVIEW VI can invoke RT target VIs remotely. The LabVIEW VI can pass parameter values to and from the RT target VIs, creating a distributed application. One advantage to communicating using the VI Server is that the VI Server allows access to the functionality of TCP while working within the framework of LabVIEW.

SMTP

The SMTP VIs can be used to send data from a VI running on the RT target to VIs running on another computer. The SMTP VIs can send electronic mail, including attached data and files, using the Simple Mail Transfer Protocol (SMTP). The SMTP VIs cannot be used to receive information. SMTP is non-deterministic, and using SMTP communication inside a time-critical VI adds jitter to the application.

Serial
Serial communication is the transmission of data between two locations through the serial ports. The VISA functions provide serial communication support in LabVIEW for communication between RT targets with serial devices and serial instruments or computers that have a serial connection. Serial communication is ideal when transfer data rates are low or for transmitting data over long distances. Serial communication is non-deterministic, and using serial communication inside a time-critical VI adds jitter to the application.

IrDA wireless communication
Infrared Data Association (IrDA) is a communication standard that specifies a way to transfer data using a wireless infrared connection. IrDA devices communicate using infrared LEDs. IrDA devices can be used to send data in and out of VIs running on an RT target using the LabVIEW IrDA functions. RT Series controllers support Extended Systems XTNDAccess IrDA PC Adapters and ACTiSYS IR-220L+ IrDA Com-Port Serial Adapters connected to a built-in controller serial port.

CAN
The Controller Area Network (CAN) is a deterministic, multi-drop communication bus standardized as ISO 11898. Using CAN, up to 8 data bytes per frame can be transferred at a rate of up to 1 Mbit per second. Multiple RT systems can be networked using NI-CAN interface cards and NI-CAN driver software.

Reflective memory
Reflective memory is a means of sharing data between two independent systems in a deterministic manner. Reflective memory devices can be connected together using fiber optic cables to provide this deterministic network, which operates like a dual-ported memory system. When one system acquires data it writes to its local memory, which has one port that is defined as the Reflective Memory address space that in turn updates the memory of the second system over fiber optics. In a typical Reflective Memory network, all secondary systems' local memory can be updated within 700 nanoseconds.

Reflective memory was traditionally used with RTOSs and VXI; however, solutions on other platforms have been created as well. VMIC is a company that provides a Reflective memory solution for both the PXI and PCI standards. National Instruments has created a LabVIEW instrument driver for VMIC's Reflective memory devices, which can leverage off LabVIEW Real-Time and the PXI standard for deterministic communication.

4 Conclusions

LabVIEW Real-Time and NI real-time hardware targets such as PXI and Compact FieldPoint leverage the powerful graphical programming and measurement capabilities of LabVIEW for real-time control applications. For the

design of embedded control systems, LabVIEW can be used to analyze dynamic systems and design and simulate control logic. LabVIEW Real-Time can be used to rapidly prototype the control logic and test against the real plant system by interfacing to sensor and actuator signals. In addition, LabVIEW Real-Time can be used to test a designed embedded controller in an HIL test configuration. For industrial control applications, LabVIEW Real-Time provides a single software development tool that can implement the functions of both traditional PLCs and PC-based control systems, thus acting as a PAC. Real-time hardware systems such as Compact FieldPoint provide a small, rugged form factor with the I/O capabilities necessary for distributed industrial control applications.

Acknowledgements

LabVIEW, CompactRIO, DataSocket, FieldPoint, MXI, and NI-CAN are trademarks of National Instruments. Real-Time Workshop and Simulink are registered trademarks of The MathWorks, Inc. Other product and company names listed are trademarks or trade names of their respective companies.

References

1. J. Travis, *Internet Applications with LabVIEW.* Prentice-Hall PTR, Upper Saddle River, NJ, 2000.
2. R. H. Bishop, *Learning With LabVIEW 7 Express.* Pearson Education, Inc., Upper Saddle River, NJ, 2004.
3. R. Jamal and H. Pichlik, *LabVIEW Applications and Solutions.* Prentice-Hall PTR, Upper Saddle River, NJ, 1999.
4. G. Johnson, *LabVIEW Graphical Programming.* McGraw-Hill, New York, 1997.
5. G. Johnson, *LabVIEW Power Programming.* McGraw-Hill, New York, 1998.
6. *LabVIEW Real-Time Module User Manual.* National Instruments Corporation, Austin, TX, 2004.
7. *LabVIEW Real-Time Architecture and Good Programming Practices.* http://zone.ni.com/devzone/conceptd.nsf/webmain/DC6EE13F8612FC4686256B510066724F
8. *ARC Advisory Group* http://www.arcweb.com

Control Loops in RTLinux

Victor Yodaiken, Matt Sherer, and Edgar Hilton

FSMLabs Inc., Socorro, NM 87801, U.S.A.
{yodaiken,sherer,efhilton}@fsmlabs.com

1 Control Loops in RTLinux

Control loops have the schematic form `while(active){wait for trigger; do something to a device;}` and they are the building blocks of most real-time programs. Control loops in RTLinux can be just a few lines of code or can involve the synchronized performance of thousands of loops. People have used RTLinux for some of the most demanding real-time applications as well as for quick experiments. Whether you are developing a 100-microsecond duty cycle magnetic bearing controller [3], a jet engine control and hardware-in-loop simulation (as did Pratt & Whitney), or a simple robotic controller using a sound card as an improvised analog-to-digital (A/D) device (several Japanese universities), you need the same ingredients, the same principles, and a good understanding of the device or plant you are controlling.

RTLinux consists of a real-time kernel called RTCore which runs the Linux operating system as a preemptible thread (it is also possible to use the Berkeley Software Design BSD UNIX instead of Linux). The execution of Linux software is scheduled by the Linux scheduler, independently of the RTCore scheduler, and the Linux software is prevented from disabling interrupts or timers that can trigger real-time software activation. The idea here is to use Linux as a utility task and to *decouple* real-time components from the non-real-time operating system utility. The general rule for RTLinux programmers is to move as much of the code as possible into the Linux context in order to take advantage of the sophisticated environment and tools available there so that in the real-time environment we can focus on getting the timing right.

This chapter covers basic control loop design and emphasizes issues of moving data and control information between the real-time loop and the outside world. We provide abbreviated treatment of scheduling and synchroniza-

tion. Our experience is that pure priority scheduling[1] satisfies 90% of all requirements and slot schedulers[2] satisfy most of the rest. Both are built into RTLinux. Synchronization is not hard if a few basic rules are followed [8] and proper attention is paid to making the design robust. The RTLinux programming model is most effective when engineers can properly modularize their programs and reuse the powerful software found in Linux or BSD.

RTCore uses the Portable Operating System Interface (POSIX) threads application programming interface (API) with some additions for the two-kernel design. A complete manual comes with RTLinuxPro or you can download the Single UNIX Specification from the Open Group [6] or use the POSIX specification itself [4]. The standard textbook on POSIX threads is [1]. Readers who need a quick introduction to "C" have many choices: the authoritative reference is [5]. Our examples are all for RTLinuxPro because it is simpler to use and has some additional control features, but most of what is written here is also applicable to RTLinuxFree.

The next section covers the mechanics of interrupt-driven and periodic control loops in RTLinux and how to interface these loops to external software components. Section 3 covers a control system for a servomotor-based device. Section 4 shows how control loops can be written using the XML-RPC Controls Kit tool in RTLinux to reduce interface complexity.

Among the important issues we have *not* covered are real-time networking, RTCore's memory-protected mode where threads are created inside UNIX processes instead of in the non-real-time kernel, debugging, performance analysis, shared memory, and complex synchronization. A wider introduction to RTLinux programming can be found in the RTLinux Handbook [7].

2 Basic Control Loops: Event Driven and Periodic

RTLinux applications consist of threads, interrupt handlers, "main" kernel processes, and user processes. A thread is a stream of execution—a stack, local variables, and code. An interrupt handler is a function that is called when an interrupt is caught. A process is a thread that runs under control of the UNIX scheduler. A kernel process runs in the kernel address space. A user process executes in its own restricted address space. Threads and interrupt handlers execute under the control of the real-time kernel. User code is executed by ordinary UNIX processes. Processes are threads with more state—user processes in UNIX usually have their own memory space. Interrupt handlers have the highest priority and lowest overhead but are more limited in functionality.

[1] The scheduler always selects the highest priority runnable thread and lets it run to completion until the thread blocks, or until a higher priority thread becomes runnable.

[2] The scheduler runs at fixed intervals and picks the highest priority runnable thread associated with the current interval.

RTLinux applications are written as if they were ordinary UNIX programs. A build system creates an executable file that sets up standard input and output and automatically starts the "main routine" running as a Linux kernel thread in a specialized environment. So we start with a program, say, `prog.c` and build an executable by typing `make` to produce an executable file called `prog.rtl`. When we execute `prog.rtl` the `main` function is exported to run in kernel space and any interrupt handlers or threads created by `main` run under the control of the real-time scheduler—invisible to the UNIX (Linux or BSD) platform.

2.1 Loops

Interrupt-driven loops

Table 1 is a complete application that creates an interrupt handler for a simple device. The loop is implicit and has the form `while(NOT terminated){wait for interrupt; do handler}`. Unlike a simple polling loop, however, this loop allows the processor to continue to do useful work between interrupts. The main routine registers the handler to catch some specific interrupt and then calls `rtl_main_wait` to wait for program termination. To clean up, the main routine unregisters the handler. The handler itself does whatever it needs to do to control the device, re-enables interrupts from that device (RTLinux leaves the interrupt blocked until told otherwise), and returns.

Interrupt latency. Because RTLinux is a *hard* real-time system, only other real-time software can delay the execution of an interrupt handler. On a typical IA32 PC currently (2004), the *worst-case interrupt latency* between assertion of a hardware interrupt and start of execution of the handler is about 8 microseconds.

Interrupt handlers need to be very simple. When you have a complex requirement for an interrupt-driven control loop you can use an interrupt-driven *thread* with a semaphore to signal between the interrupt handler and the thread. Let's first look at threads and then return to this issue.

Periodic control loops

Table 2 is a complete application that creates a periodic control thread for a simple device. The loop has the form `while(NOT terminated){wait for period; do mythread}`. Again, this loop allows the processor to continue to do useful work between time intervals. The main routine creates a thread using the POSIX `pthread_create` function and then waits for termination. To clean up, the main routine cancels the thread and then uses the POSIX `pthread_join` to wait for the thread to completely exit. The thread itself first reads the current time and then enters a loop where it continually waits "`period`" nanoseconds and then does whatever it needs to do in terms of control. The POSIX standard allows for several clocks with different properties.

```
#include <stdio.h>
#include <unistd.h>
#include <sched.h>

unsigned int handler(unsigned int irq, struct rtl_frame *regs);

int main(void) {
  if ( rtl_request_irq( IRQ, handler )) {
    printf("failed to get irq %d\n", IRQ);
    return -1;
  }
  rtl_main_wait();
  rtl_free_irq(IRQ);
  return 1;
}
unsigned int handler(unsigned int irq, struct rtl_frame *regs)
{
  CONTROL_DEVICE();
  rtl_hard_enable_irq(irq);
  return 0;
}
```

Table 1. Interrupt-driven control loop

`CLOCK_REALTIME` is the default in RTLinux, but `CLOCK_GPOS` which is phase lock looped to the timer in the Linux (or BSD) operating system, `CLOCK_GPS` which can be phase lock looped to a GPS clock, and `CLOCK_MONOTONIC` which is the time since system boot are also sometimes useful. Usually periodic control loops are on an "absolute" schedule so we use the `TIMER_ABSTIME`, however we could also wait for a time relative to the current time. The timers are, of course, not completely precise and a periodic task has a certain level of *jitter* as hardware limits and possibly other real-time code can introduce delays. On a standard (2004 time period) 2GHz processor, hardware induced worst-case jitter is about 22 microseconds. If that number is too high, there is a timer advance feature that expires the timer early and "busy waits" for the right moment, and even a method of reserving processors on a multiprocessor system for real-time activity only.

Note: One common error to avoid is to convert an absolute schedule to a relative schedule by moving the `rtl_clock_gettime` inside the loop body in a loop like the one in Table 2. As a result, instead of waiting for time $start + n * period$ at the nth iteration of the loop, we wait for time $start + n * period + \Sigma_{i=1}^{n} jitter_n$ as the jitter errors accumulate.

```
#include <stdio.h>
#include <pthread.h>
#include <unistd.h>

pthread_t thread;
void *mythread(void *);

int main(void)
{
  pthread_create( &thread, NULL, mythread, (void *)0 );
  rtl_main_wait();
  pthread_cancel( thread );
  pthread_join( thread, NULL );
  return 0;
}
void *mythread(void *t)
{
  struct timespec next;
  rtl_clock_gettime( CLOCK_REALTIME, &next);
  while (1) {
    timespec_add_ns( &next, period);
    clock_nanosleep( CLOCK_REALTIME, TIMER_ABSTIME,
             &next, NULL);
    CONTROL_DEVICE();
  }
    return NULL;
}
```

Table 2. Periodic control loop

Using a thread with an interrupt handler

A thread can be used in an event-driven loop if it is driven by a semaphore instead of a hardware interrupt. In the example given in Table 3, a handler catches an interrupt, reads data from the device and writes it to a *FIFO*, and then signals a thread to do the more complex processing. RTLinux FIFO connect real-time threads to each other or to non-real-time processes. The name "FIFO" means *first-in first-out* and the FIFO provides character stream connection. The main routine is responsible for creating and shutting down the thread (canceling and joining) and closing and unlinking the FIFOs.

2.2 Connecting loops to the outside world

Suppose we want to collect data from the device and store it, or we need to be able to have an operator or a control program modify the period of a loop. RTLinux provides three options: FIFOs, shared memory, and variable export. The simplest is a FIFO. When an ".rtl" program starts, it sets up

```
/* in a real program you should test for errors on
   the calls to create and open FIFOs, and to create the
   thread */
rtl_sem_t sem;
unsigned int handler(unsigned int irq, struct rtl_frame *regs);
void *thread_func(void *);
int fd_in,fd_out;
main(){
  rtl_request_irq( IRQ, handler ));
  mkfifo("/myfifo",0);
  fd_in = open("/myfifo",O_RDONLY | O_NONBLOCK);
  fd_out = open("/myfifo",O_WRONLY | O_NONBLOCK);
  pthread_create(&mythread,NULL,thread_funci, (void *)0);
  rtl_main_wait();
  rtl_free_irq(IRQ);
  pthread_cancel(mythread);
  pthread_join(mythread);
  return 0;
}
void handler(unsigned int irq, struct rtl_frame *regs){
  GET_DATA(&buffer);
  write(fd_out,&buffer,SIZE);
  rtl_hard_enable_irq(IRQ);
  rtl_sem_post(&sem);
}
void *thread_func(void *x){
 while(1){
    rtl_sem_wait(&sem);
    read(fd_in,&buffer2,SIZE);
    DO_PROCESSING;
    }
}
```

Table 3. Semaphores and FIFOS in interrupt-driven loop

standard output and standard input through real-time FIFOs. Suppose the `CONTROL_DEVICE` function used above looked like this:

```
CONTROL_DEVICE(){
  int i = READ_INPUT();
  rtl_printf("%d\n",i);
  }
```

Then the shell command `prog.rtl > logfile` would log data. The advantages of a Linux/UNIX platform become obvious here as we can use pipes to pass the data through processing programs or to graphics programs or even over the network. If we have a Linux program called `myfilter` to process the data and then want to send it over a network we can use a single line shell

command: `prog.rtl |myfilter | netcat -p 1001 remote.fsmlabs.com`. More sophisticated uses of FIFOs are discussed in the following sections.

3 A servocontroller

Suppose we have two servomotors controlled via a digital I/O port and that we also need to be able to send the control system commands from a human operator. This example, taken from a paper by Cort Dougan [2] on remote control of a video camera, shows how to control servomotors, how to accept commands from an operator, and how to work with multiple threads.

3.1 The control loops

The computer controls servomotors by varying the length of a pulse sent every cycle. The loop executes every `DutyCycle` time units, give or take some error. Each time the loop runs, an output signal gets turned on for `PulseWidth` time units and then the loop waits `DutyCycle` − `PulseWidth` time units to pass before running again. But we have two motors. Many real-time applications and even real-time operating systems have been written as big control loops with slots for operations and delays between slots. In this case, we would need to turn on both output signals, wait the minimum of the two delays, turn off the corresponding signal, wait for the difference (unless the difference is below a threshold), and then turn off the second signal. If we were implementing this loop in the context of a dedicated controller, then this approach would be ugly but reasonable, although obviously each additional motor makes things geometrically more complex—imagine six motors. But we need the processor to be free to do other work, so our control loop would have to also run the non-real-time software sometimes. By using separate threads and timers for each control loop, we can dispense with the entire problem. We don't even have to bother to put the threads on out of phase schedules or worry about priority because the worst case is when one thread must wait for the other to finish inverting the output bit—less than 2 microseconds measured on low-end PCs.

The code for the control loop can be placed in a thread. In fact we can use the same code for both threads, just passing the motor identification number to each thread (see Table 4).

3.2 The interface

Now let's consider the hard part: an operator interface. We'll start with something very primitive and take advantage of the flexibility of the UNIX design to make it smarter. In our `main` code we can create two *FIFOs* and attach a handler to them. Call one "/horizontal-motor" and call the other one "/vertical-motor". These handlers become activated on write operations from the user

```
void *servo_thread(void *argument){
  int motor_id = (int)argument; /* passed when thread is created*/
  rtl_clock_gettime(RTL_CLOCK_REALTIME,&next);
  while(active()){
     pulse_length= input_pulse_length[motor_id];
     ioctl(fd_par, RTL_PAR_SETBIT, num);
     timespec_add_ns( &next, pulse_length); /*now + pulse_length*/
     clock_nanosleep( CLOCK_REALTIME, TIMER_ABSTIME, &next, NULL);
     ioctl(fd_par, RTL_PAR_CLEARBIT, num);
     timespec_add_ns(&next, frame_length - pulse_length);
                                        /*remaining time*/
     clock_nanosleep( CLOCK_REALTIME, TIMER_ABSTIME, &next, NULL);
  }
}
```

Table 4. Control loop for a servomotor

side. That is, when the user writes a command into a FIFO, the FIFO handler runs. The idea is that the threads control the device and user commands come up the FIFOs, triggering the FIFO handlers, which then read commands and update the value of `input_pulse_length` parameters for the appropriate control loop. Commands can be typed from the shell prompt.

```
echo 0 > /horizontalmotor
echo 180 > /horizontalmotor
```

will shift our camera left and then right while

```
echo 0 > /verticalmotor
echo 180 > /verticalmotor
```

will shift it up and down. Because of the modular UNIX design, we can pretty easily connect these FIFOs to a web server and use a browser as a control interface. In fact, one of the motivations for the design of RTLinux was to take advantage of the scripting languages, browsers, and other utilities in Linux instead of being forced to use some variant of those utilities from a specialized operating system. Notice that because all the user applications run in the application operating system thread, they cannot interfere with the timing of the real-time threads!

We will not give the details of the program here—it's standard POSIX. The outline of the program is as follows.

```
int main(void) {
int error, VERTICAL,HORIZONTAL;
struct rtl_sigaction act;

/* create the FIFOs */
... rtl_mkfifo ...
```

```
/* open the FIFOs */
... rtl_open ...
/* attach FIFO handlers */
... rtl_sigaction ...
/* create threads */
... rtl_pthread_create ...

/* wait for the program to be killed */
rtl_main_wait();

/* cleanup */
         ....
```

3.3 More advanced considerations

A tougher application might require a little more work on synchronization and priorities. If the duty cycle required more precision, we could easily start the second thread on a time offset so that it was out of phase with the first loop. For even more precision, we could use a symmetric multiprocessor (SMP) machine and put the two threads on different processors or even reserve a processor for the motor control. Both of these are done via the POSIX *thread attributes* which are set prior to creating a thread. Finally, we might need additional real-time threads to, for example, collect data from the frame buffer. In this case, since the motor control threads are fast, a likely design is to give the frame buffer loop lower priority than the motor control loops so that it won't interfere with their scheduling. All these bring up interesting issues of synchronization (see [8] for some ideas on that topic). Our experience with RTLinux applications is that synchronization is a problem solved by keeping to clear rules, so we'll return to the operator interfaces. In practice, these interfaces are often the hardest things to get right.

4 Controls Kit

(This section is taken from a paper by Edgar Hilton as adapted in the RTLinux Handbook [7]). The purpose of Controls Kit (CKit) is to automate the process of connecting a control system to the outside world. CKit provides methods to monitor any parameter, tune parameters, or listen to alarms (both locally and/or through the network) from any

1. script,
2. web browser,
3. spreadsheet,
4. XML aware program,
5. XML-RPC aware interface,

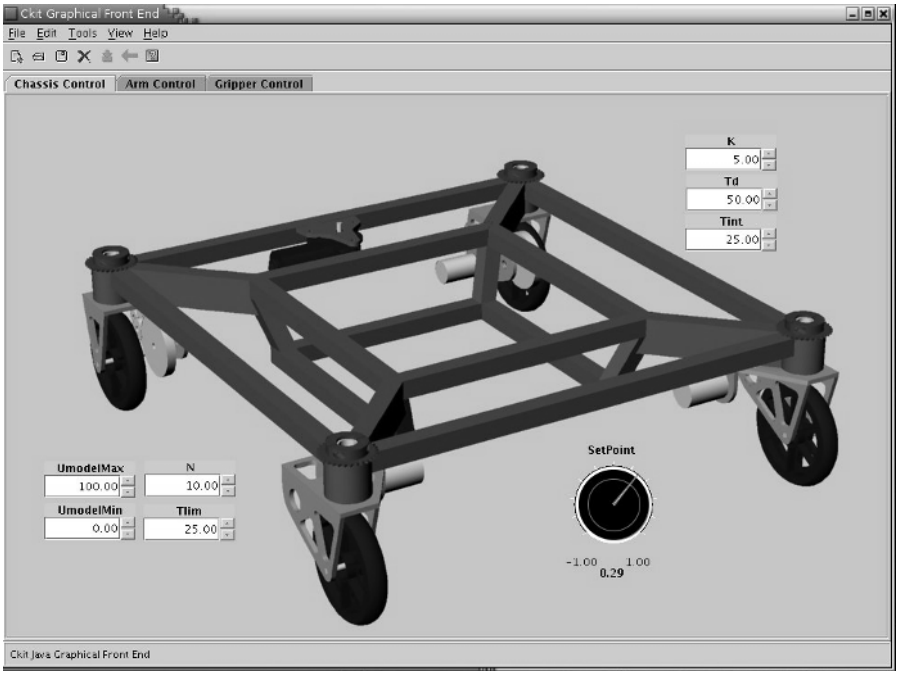


Fig. 1. Detail of CKit Java GUI

6. provided Java based graphical user interface (GUI), as shown in Fig. 1.

A CKit enabled RTLinuxPro box can be monitored not only by a remote engineering machine, but also by other CKit enabled RTLinuxPro boxes over the network![3]

The example presented is a single-input single-output (SISO) proper, anti-windup, and high frequency limited proportional-integral-derivative (PID) controller using the algorithm provided in the FSMLabs controls library with CKit (see Fig. 2). In this example, we'll assume that there is an existing device driver for the A/D and D/A hardware.

4.1 Design

We only need one real-time thread in this example. Initialization of the hardware and the PID parameters and running the trigger function (to send a command RUN or STOP to the thread) all can be done without real-time. The loop period is 1/2 millisecond (500 microseconds). When the thread is

[3]Imagine, for example, a dozen mobile robots synchronizing with each other via a wireless network or a set of factory floor control workstations synchronizing part arrivals and departures with each other while at the same time being tuned remotely from either an engineering or management safe house.

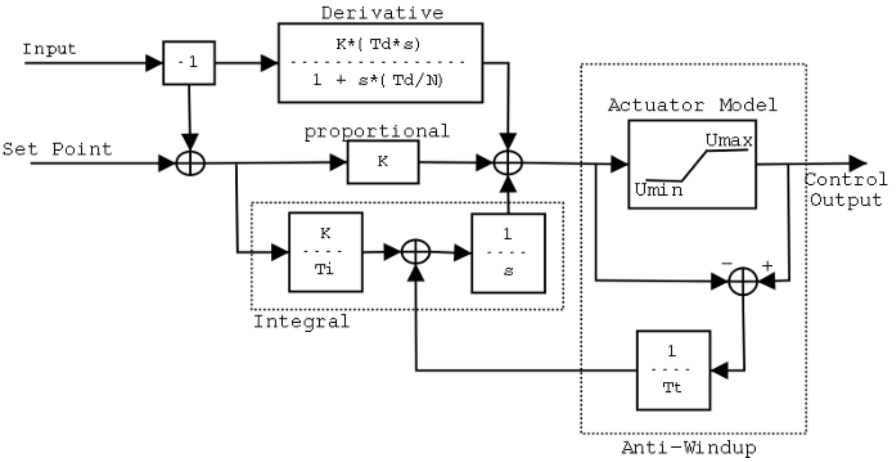


Fig. 2. Detail of PID algorithm

created, it will suspend itself on a semaphore waiting for a RUN command. The RUN command will release the thread to run a periodic control loop until a STOP command is issued.

4.2 Implementation

CKit controlled objects are called *entities*, and in this program we need three entities.

1. `TestGroup` is the top-level entity of type "group" and it serves as the root of the entity tree.
2. `Run` is a boolean entity which we will associate with a handler function so that any changes to this entity will trigger the handler function.
3. `myPID` is an algorithm entity defined within the FSMLabs controls library (`libFSMCL.a`). This entity contains all the parameters and internal states needed for the PID algorithm.

4.3 Coding

The example structure is straightforward:

```
import headers and declare globals and CKit variables
main(){
  initialize CKit variables
  create thread
  rtl_main_wait()
  cleanup
  exit(0);
```

```
}
thread() { ... }
trigger(){ ... } /* the FIFO event handler */
```

First, the headers, globals, and CKit variable declarations are shown in Table 5. Here are the declarations of the CKit aggregate entity `MainPID`, basic entity `RunBool`, and basic entity `TestGroup`.

```
#include <rtl_posixio.h>
#include <stdio.h>
#include <ckit/FSMCL_core.h>
#include <semaphore.h>

/* Declarations */
rtl_sem_t loop_sem;
FSMCL_PID_entity MainPID;  /* PID  entity from library */
CK_entity RunBool;         /* boolean entity */
CK_entity TestGroup;       /* group entity */

/* Function prototypes */
static void *controlloop (void *arg);
static void triggerFcn(void *arg);
```

Table 5. CKit header section

The variable "TestGroup" is registered as the top-level directory by giving it a "NULL" parent. We give it an unimaginative name "Test" and a tool-tip of "Test Group for PID Example". The code for this is shown in Table 6. The boolean can be initialized to FALSE, assigned as a child of the `TestGroup` entity, and associated with the function "`triggerFcn()`" so that the next time that this entity is updated by the user the given function will run in the context of the Linux thread. All the parameters to the PID controller are statically initialized in this example, but we could have done the same thing from the command line through a script prior to running.

The control loop thread can then be created using the usual RTLinux API calls. When `pthread_create` executes, the real-time application launches.

Finally, we have to make sure to do a cleanup and release all resources after the `rtl_main_wait`. Cleanup closes down the control loop thread and then destroys all of the entities. Destroying the top-level entity automatically frees all of its children.

Trigger function

As you may recall, each time that the "Run" entity is updated from the command line, the handler function will execute and either start or stop the

```
main()
{
 pthread_t testThrd;

 /* initialize our entities */
 CK_group_init(&TestGroup,                    /* entity pointer */
               "Test",                        /* registration name */
               "Test Group for PID Example", /* tooltip */
                NULL);                        /* parenthood */
 FSMCL_PID_init(&MainPID,                     /* entity pointer */
                "PIDControl",                 /* registration name */
                "Test PID controller",        /* tooltip */
                &TestGroup);                  /* parenthood */
 CK_boolean_init(&RunBool,                    /* entity pointer */
                 "Run",                       /* registration name */
                 "Run/Stop program",          /* tooltip */
                 &TestGroup,                  /* parenthood */
                 FALSE);                    /* initial value */
 CK_execute_on_update(&RunBool,               /* entity pointer */
                      triggerFcn,             /* pointer to handler */
                      NULL);                  /* argument to handler */

 /* Initialize the PID controller gains (min, max, initial) */
 CK_scalar_init_float_val(&(MainPID.K),    0.0,   10.0,  5.0);
 CK_scalar_init_float_val(&(MainPID.Td),   10.0, 100.0, 50.0);
 CK_scalar_init_float_val(&(MainPID.Tint), 1.0,  100.0, 25.0);

 /* anti-windup and high frequency limit (min, max, initial)*/
 CK_scalar_init_float_val(&(MainPID.Tlim),  1.0, 100.0, 25.0);
 CK_scalar_init_float_val(&(MainPID.N),     3.0,  20.0, 10.0);

 /* Saturation model for actuator (min, max, initial) */
 CK_scalar_init_float_val(&(MainPID.Umodel.UmodelMin),
  -10.0,  10.0,   0.0);
 CK_scalar_init_float_val(&(MainPID.Umodel.UmodelMax),
   90.0, 110.0, 100.0);

 /* And our reference point (min, max, initial) */
 CK_scalar_init_float_val(&(MainPID.SetPoint),  -1.0, 1.0, 0.0);

 pthread_create(&testThrd, NULL, controlloop, 0);

 rtl_main_wait();
 pthread_cancel(testThrd);
 pthread_join(test, NULL);

 /* Cleanup our entities */
 CK_entity_destroy(&TestGroup);
 exit(0); /* end the main routine */
}
```

Table 6. CKit main routine

program based on the value of the boolean. The entire trigger function is given in Table 7.

```
static void triggerFcn(void *arg)
{
  int val;
  val = CK_scalar_get_boolean(&RunBool);

  if (val){
    CK_message("Commencing control run");
    rtl_sem_post(&loop_start);
  } else {
    CK_message("Controller shut down by user");
  }
}
```

Table 7. CKit trigger function

In this example, we will generate an alarm of level 0 (e.g., a low-priority message) through the CKit infrastructure each time the boolean is updated. If the program is to be started, then we will kick-start the control loop thread. Otherwise, we simply enqueue the alarm and let the control loop thread shut itself down after the next iteration.

Control loop

Our final coding concern is the design of the control loop thread. Here, we set up two while loops. The innermost loop is nothing more than a standard periodic loop as seen in many of the RTLinux example programs. The external while loop is an infinite loop which will be used to run and stop our program.

The outermost loop will first check the value of the runtime boolean. If the value is `false`, it will sleep on the semaphore until it is awakened by the trigger function (which will post the semaphore). At that point, it will initialize the internal values of the PID controller (`FSMCL_PID_reset()`)and will begin execution of the periodic component of the thread. Within the periodic component, we will sample our A/D boards using some previously existing I/O function `SampleAD()`, calculate the PID control using the sensed values (`FSMCL_PIDcontrol()`), and finally write out the control output through our D/A boards through a previously existing function `WriteDA()`.

The periodic component will continue to execute until the runtime boolean (`RunBool`) becomes `false`. As soon as it becomes `false`, the code will exit the innermost while loop. Then, prior to once again suspending itself, the code will convert the thread into a non-periodic thread.

The entire code for the main thread is shown in Table 8.

```
static void *controlloop (void *arg)
{
  struct timespec next;
  float Sens;        /* Sensor value */
  float Control;     /* Control output */
  float Period;      /* PID controller period. Needed by
                        the PID function */

  Period = 0.05;      /* Desired Period, 500 microseconds */

  pthread_setfp_np(pthread_self(),1);

  while (1) {
    if (!CK_scalar_get_boolean(&RunBool)){
      rtl_sem_wait(&loop_start);
    }
    FSMCL_PID_reset(&MainPID);
    rtl_clock_gettime( CLOCK_REALTIME, &next);

    /* Periodic loop */
    while(CK_scalar_get_boolean(&RunBool)){
      timespec_add_ns( &next, (int)(Period*1.0e9));
      clock_nanosleep( CLOCK_REALTIME, TIMER_ABSTIME, &next, NULL);

      // sample sensors from A/D
      Sens = SampleAD();

      // Run PID controller
      FSMCL_PIDcontrol(&Sens, &Control, &Period, &MainPID);

      // write out to D/A
      WriteDA(&Control);
    }
  }
  return 0;
}
```

Table 8. CKit example

4.4 Program execution

We reuse our RTLinuxPro Makefile as above to build our CKit aware RTCore program. We execute our program as follows.

1. RTCore must be up and running.
2. The hard real-time and non-real-time components of CKit can be started from the command line with the commands `ckit_module.rtl` and `ckitd`, respectively.

3. If we want to allow remote machines to query and set entities, we start up the XML-RPC server with the command `ck_xmlrpc_server`.
4. Execute the new CKit aware program (`./test.rtl`)

4.5 Parameter queries

We can now take a look at the program variables by typing:

```
ck_hrt_op -L
```

which will return the output shown in Table 9.

```
+-#>TestGroup          #  group  #
| +-#> PIDControl      #  group  #
| | +-#> ActuatorModel #  group  #
| | | +-#> UmodelMax   #  float  #  100.00
| | | +-#> UmodelMin   #  float  #   0.00
| | |
| | +-#> K             #  float  #   5.00
| | +-#> N             #  float  #  10.00
| | +-#> SetPoint      #  float  #   0.00
| | +-#> Td            #  float  #  50.00
| | +-#> Tint          #  float  #  25.00
| | +-#> Tlim          #  float  #  25.00
| |
| +-#> Run             # boolean #  false
|
```

Table 9. CKit human readable tree

Here, our registered tree is shown with the basic entities `TestGroup` and `RunBool`, as well as all of the entities defined within the `PIDControl` entity. An XML rendition of the same tree is obtained by typing:

```
ck_hrt_op -Lx
```

We can embed the `ck_hrt_op` utility into any scripting language that can interpret XML (such as Perl, Python, etc.). But to simplify the explanation, we will stick to the command line here. The program can be started by typing:

```
ck_hrt_op -p TestGroup::Run -s true -v
```

where the `-p` option is used to specify the name of the entity, `-s` is used to specify the value to which we are setting the entity, and `-v` specifies which value we want to specify, in this case, the current value (as compared to the minimum or maximum values). Once we type this command, the program should immediately begin its execution. To stop the program, type:

```
ck_hrt_op -p TestGroup::Run -s false -v
```

To verify that the `Run` flag has been updated, we could type:

```
ck_hrt_op -p TestGroup::Run -g -v
```

which states that we want to obtain (`-g`) the current value (`-v`) of the `Run` entity (`-p TestGroup::Run`). This returns the string value of "false".

We can also query the values and set the values for remote CKit enabled machines, assuming that the remote machine is already running the `ck_xmlrpc_server`. You can do so by using the `-X` option in `ck_hrt_op` as follows:

```
ck_hrt_op -X http://remoteMachineIP:3134/RPC2 -L
```

where `remoteMachineIP` denotes the address of the remote CKit enabled machine, 3134 denotes the port number (by default, port 3134, configurable), and `RPC2` denotes that this is the XML-RPC protocol.

All of the above `ck_hrt_op` capabilities are also accessible via a C++ library which can perform both local and remote parameter queries. This allows you to write your own CKit aware C++ programs which directly query the `ckitd` server. If, however, you prefer to query the remote machines via a different programming language (such as Perl), you can also do so by performing XML-RPC queries to the remote machine. Refer to the CKit manual and examples for a more complete discussion of these subjects.

4.6 Subscribing to asynchronous alarms

At this point, the reader will have noticed that each time that the value of the `Run` boolean changed, we should have received an asynchronous alarm. What happened to these alarms?

The CKit infrastructure allows any number of users to subscribe to any alarm level (0–10) that may be especially beneficial to the users. To do so, we rely on either the `ck_alarm` command line utility or the C++ libraries described at the closing of the previous section.

For example, let's say that we are interested in subscribing to all alarms being generated within the CKit. To do so, we type the following:

```
ck_alarm -s all
```

Here, we are going to print out a human readable alarm each time that an alarm occurs. In this case, we are listening to all alarm levels, normal low priority messages (level 0), warnings (level 5), and critical alarms (level 10). We could have listed the explicit alarm levels of interest on a comma-separated list, as in

```
ck_alarm -s 0,5,10
```

which will subscribe to all normal messages, warnings, and critical messages, respectively. To unsubscribe, simply hit Control+C, and the utility will unsubscribe from all the alarms.

If needed, we can take specialized actions on different alarms of different levels. All alarms can be displayed either in human readable or XML format.

4.7 Graphical interfaces

The CKit is currently capable of interfacing to many different programming languages due to its XML infrastructure, but it also comes with a Java front end which can be used for many control applications. Fig. 1 shows one such front end for a robotic application. Three tabs are shown, one which is used to control the robotic chassis, a second one which is used to tune the parameters of the robotic arm, and a third one used to control the robotic gripper. Entities are implemented as widgets which can be used to update and monitor entity values. These widgets can further be placed above any graphic which can more easily document the control project.

References

1. D.R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, Reading, MA, 1997.
2. Cort Dougan. Two-axis, real-time camera control. *Dr. Dobbs Journal*, October 2002.
3. Marty Humphrey, Edgar Hilton, and Paul Allaire. Experiences using RT-linux to implement a controller for a high speed magnetic bearing system. In *IEEE Real Time Technology and Applications Symposium*, pages 121–130, 1999.
4. Information Technology—Portable Operating Systems Interface (POSIX)—Part 1: System Application Program Interface (API)—Amendment 2: Threads Extension [C Language]. IEEE Standard 1003.1c–1995, IEEE, New York, 1995. Also ISO/IEC 9945-1:1990b.
5. B.W. Kernighan and D.M. Ritchie. *The C Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1988. 2nd edition.
6. OpenGroup. The Single UNIX Specification Version. `http://www.unix.org/version3/ieee_std.html` Catalog number C046.
7. Matt Sherer and FSMLabs Technical Staff. *Real-Time Programming in RTCore*. FSMLabs, Socorro, NM, 2004. Versions with each release.
8. Victor Yodaiken. Temporal inventory and real-time sychronization in rtlinuxpro. Technical report, FSMLabs, Socorro, NM, April 2003.

Part IV

Theory

An Introduction to Hybrid Automata

Jean-François Raskin

Computer Science Department
University of Brussels
Belgium
jraskin@ulb.ac.be

1 Introduction

Hybrid systems are digital real-time systems embedded in analog environments. A paradigmatic example of a hybrid system is a digital embedded control program for an analog plant environment, like a furnace or an airplane: the controller state moves discretely between control modes, and in each control mode, the plant state evolves continuously according to physical laws. Those systems combine discrete and continuous dynamics. Those aspects have been studied in computer science and in control theory. Computer scientists have introduced *hybrid automata* [28], a formal model that combines discrete control graphs, usually called *finite state automata*, with continuously evolving variables. A hybrid automaton exhibits two kinds of state changes: *discrete jump transitions* occur instantaneously, and *continuous flow transitions* occur when time elapses.

Hybrid systems are often systems that are safety critical. As a consequence, their reliability is a central issue. For example, the correctness of a digital controller that monitors the temperature of a nuclear reactor is crucial. We present hybrid automata as formal models that define *trajectories* (behaviors) of hybrid systems. Properties of a hybrid system assign values to its trajectories: for example, they can classify trajectories as good or bad. The behaviors of a hybrid automaton are often complex, and thus it may be difficult to reason about them. This is why, since the early works on hybrid automata, the emphasis has been on their *computer aided analysis*. Model-checking methods [18] have been studied extensively and tools able to analyze complex hybrid systems have been developed.

This chapter is organized as follows. First, we introduce the *syntax* and *semantics* of hybrid automata, and show how complex hybrid systems can be modeled *compositionally* as *products of* hybrid automata. Then, we define *safety properties* of hybrid automata and show how to model them using *monitors*. We show that the *verification* of those properties reduces naturally to reachability problems, that is, to decide if there exists a trajectory of the

hybrid system that reaches a given set of states. As hybrid automata can be very complex mathematical objects, restricted subclasses for which we have *automatic analysis methods* have been introduced. In this introduction, we focus on *rectangular hybrid automata* and show how they can be used to *over-approximate* the behavior of more complex hybrid automata. We close the chapter by referencing the literature to allow the reader into go deeper into this flourishing research subject.

2 Hybrid Automata: A Model for Hybrid Systems

To illustrate the main notions about hybrid automata, we use a running example throughout the chapter. The components of the running example are depicted in Fig. 1. It shows a system composed of three devices: (i) a tank that contains water and that can be heated using a gas burner, (ii) a gas burner that can be *turned on* or *turned off*, and (iii) a thermometer that monitors the temperature of the water inside the tank and periodically issues signals when the temperature of the water in the tank is above or below certain thresholds. Later, we will add to this system a *controller* that will observe the signals issued by the thermometer and will issue orders to the gas burner in order to maintain the temperature of the water within a given range.

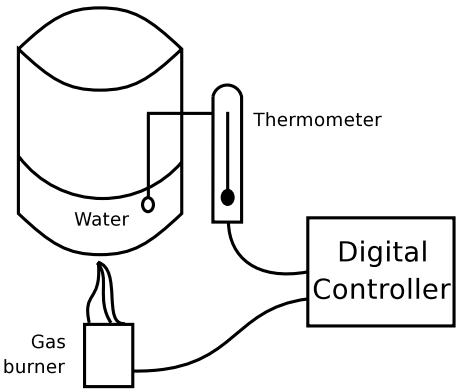


Fig. 1. Our running example

We first describe in detail the behavior of the temperature of the water in the tank. When the gas burner is OFF, the temperature of the water, denoted by the variable x, decreases according to the following exponential function: $x(t) = Ie^{-Kt}$ where I is the initial temperature of the water, K is a constant that depends on the nature of the tank (how much it conducts heat for example), and t denotes time. However, this law is only true when the

temperature of the water is greater than 20 degrees, the temperature of the room where the tank is located. When the heater is OFF and the temperature of the water is 20 degrees, then the temperature stays constant. On the other hand, when the gas burner is ON, the temperature of the water increases according to the following exponential function $x(t) = Ie^{-Kt} + h(1 - e^{-Kt})$ where I, K, and t are as before and h is a constant that depends on the power of the gas burner. Again, this rule is only true if the water in the tank has a temperature that is less than or equal to 100 degrees. When the temperature of the water reaches 100 degrees, it stays constant (the pressure increases but we omit that in our model). Fig. 2 shows a fragment of a possible evolution of the temperature of the water within the tank.

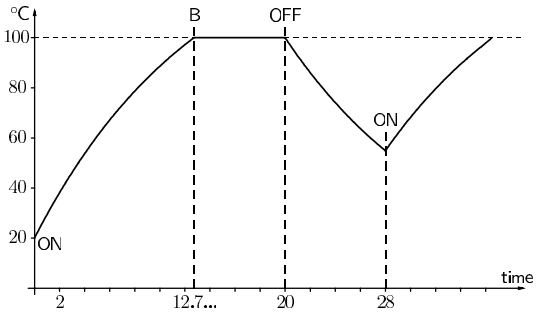


Fig. 2. One possible behavior of the tank

As we can see from the description of the evolution of the temperature in the tank, the system is not purely continuous. The evolution of the temperature depends on the mode of the system (the burner is ON or OFF, the temperature is below or above 100 degrees), and the system can switch discretely from one mode to another (if the burner is turned off, for example). Therefore, a natural model for such a system should mix continuous evolutions with discrete switches. Hybrid automata are well suited to describe such complex mixed discrete-continuous behaviors. Their syntax is defined in the next subsection.

2.1 Syntax

A hybrid automaton is a *generalized finite-state automaton* that is equipped with continuous variables. The *discrete changes* of the hybrid system are modeled by *edges* of the automaton, and the *continuous evolutions* of the hybrid system are modeled by *differential equations* that label locations of the automaton. The syntax of hybrid automata is defined as follows.

Definition 1 [Hybrid Automaton] A *hybrid automaton* H is a tuple $\langle \mathsf{Loc}, \mathsf{Edge}, \Sigma, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump} \rangle$ where:

- Loc is a finite set $\{l_1, l_2, \ldots l_n\}$ of (control) locations that represent *control modes* of the hybrid system.
- Σ is a finite set of *event names*.
- $\mathsf{Edge} \subseteq \mathsf{Loc} \times \Sigma \times \mathsf{Loc}$ is a finite set of *labelled edges* that represent *discrete changes* of control mode in the hybrid system. Those changes are labelled by *event names* taken from the finite set of labels Σ.
- X is a finite set $\{x_1, x_2, \ldots, x_m\}$ of real-valued variables. We write $\dot{X}$ for the set of dotted variables $\{\dot{x}_1, \dot{x}_2, \ldots, \dot{x}_m\}$ which are used to represent first derivatives of the variables during continuous evolutions (inside a mode), and we write X' for the primed variables $\{x'_1, x'_2, \ldots, x'_m\}$ that are used to represent updates at the conclusion of discrete changes (from one control mode to another).
- $\mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}$ are functions that assign three predicates to each location. $\mathsf{Init}(l)$ is a predicate whose free variables are from X and which states the possible valuations for those variables when the hybrid system starts from location l. $\mathsf{Inv}(l)$ is a predicate whose free variables are from X and which constrains the possible valuations for those variables when the control of the hybrid system is in location l. $\mathsf{Flow}(l)$ is a predicate whose free variables are from $X \cup \dot{X}$ and which states the possible continuous evolutions when the control of the hybrid system is in location l.
- Jump is a function that assigns to each labelled edge a predicate whose free variables are from $X \cup X'$. $\mathsf{Jump}(e)$ states when the discrete change modeled by e is possible and what the possible updates of the variables are when the hybrid system makes the discrete change.

The evolution of the temperature of the water in the tank is modeled using the hybrid automaton of Fig. 3. Locations are drawn as boxes with rounded corners and edges as arrows. Locations are named t_1 to t_4. A predicate next to a location denotes an invariant predicate. Invariant predicates equivalent to *true* are omitted. A predicate next to a location within a box denotes an initial predicate. Initial predicates equivalent to *false* are omitted. Predicates inside locations denote flow predicates. Edges are labelled by event names and update predicates. The hybrid automaton is composed of four locations that model the four different modes of evolution of the temperature within the tank as described above. Variable x is used to model the temperature of the water in the tank.

Location t_1 models the behavior of the system when the temperature of the water is between 20 and 100 degrees as indicated by the invariant $20 \leq x \leq 100$ and the gas burner is ON. In that case, the dynamics that govern the variable x is given by the flow predicate $\dot{x} = K(h - x)$. Location t_2 models the behavior of the system when the temperature of the water has reached 100 degrees. In that location, the flow predicate $\dot{x} = 0$ models the fact that the water temperature stays constant. Location t_3 models the tank system when the heater is OFF and the temperature of the water is above 20 degrees. In that case, the flow predicate that governs the evolution of x is $\dot{x} = -Kx$. Finally,

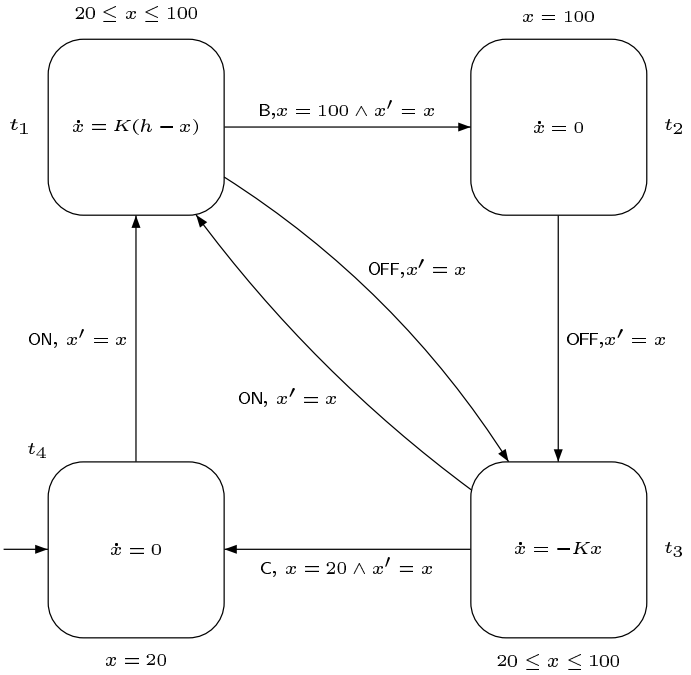


Fig. 3. A hybrid automaton for the tank

location t_4 models the behavior of the tank when the gas burner is OFF and the temperature of the water is equal to 20 degrees.

The edge from location t_1 to location t_2 is crossed when the temperature of the water reaches 100 degrees. In that situation, the control of the hybrid automaton cannot stay in location l_1 without violating the invariant $20 \leq x \leq 100$. The predicate $x = 100$ ensures that this edge can only be crossed when the temperature has reached 100 degrees. Edge from location t_1 to location t_3 is crossed when the burner is *turned off*. In that case, the dynamics of the system changes instantaneously from $\dot{x} = K(h-x)$ to $\dot{x} = -Kx$. This edge can be crossed at any time when the control is in location t_2. In the sequel, we fix the value of K to be 0.075 and h to be 150.

2.2 Semantics

At any instant, the *state* of the hybrid system specifies a *control location* and values for all *real-valued variables*. The state can change in two ways: (i) by an instantaneous transition jump that changes possibly both the control location and the values of some real-valued variables, or (ii) by a time delay that changes only the values of the real-valued variables in a smooth manner according to the flow and invariant of the current control location. To capture

those behaviors in a formal way, we use *timed transition systems*, which are defined as follows.

Definition 2 [Timed Transition System] A *timed transition system* TTS is a tuple $\langle S, S_0, \Sigma, \rightarrow \rangle$ where S is a (possibly infinite) set of states, $S_0 \subseteq S$ is the subset of initial states, Σ is a finite set of labels, and $\rightarrow \subseteq S \times \Sigma \cup \mathbb{R}^{\geq 0} \times S$ is the transition relation. We write $s \rightarrow_d s'$ for $(s, d, s') \in \rightarrow$.

We denote $[X \rightarrow \mathbb{R}]$ the set of valuations that map variables from X to real numbers. Let p be a predicate over the set of variables X, then $[\![p]\!]$ is the set of valuations $v \in [X \rightarrow \mathbb{R}]$ satisfying p, noted $v \models p$. Let q be a predicate over the set of variables $X \cup X'$, then $[\![q]\!]$ is the set of pairs of valuations $(v, v') \in [X \rightarrow \mathbb{R}] \times [X' \rightarrow \mathbb{R}]$ such that $(v, v') \models q$. Let r be a predicate over the set of variables $X \cup \dot{X}$, then $[\![r]\!]$ is the set of pairs of valuations $(v, \dot{v}) \in [X \rightarrow \mathbb{R}] \times [\dot{X} \rightarrow \mathbb{R}]$ such that $(v, \dot{v}) \models r$. The TTS associated to a hybrid automaton is defined as follows.

Definition 3 The timed transition system $\langle S, S_0, \Sigma, \rightarrow \rangle$ of the hybrid automaton $H = \langle \mathsf{Loc}, \mathsf{Edge}, \Sigma, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump} \rangle$, written as $[\![H]\!]$, is defined as follows:

- S is the set of pairs (l, v) where $l \in \mathsf{Loc}$ and $v \in [X \rightarrow \mathbb{R}]$ such that $v \in [\![\mathsf{Inv}(l)]\!]$, this set is called the *state space* of H;
- S_0 is the subset of pairs $(l, v) \in S$ such that $v \in [\![\mathsf{Init}(l)]\!]$, this set is called the *initial state space* of H;
- the transitions are either:
  - **discrete**: for each edge $e = (l, \sigma, l') \in \mathsf{Edge}$, we have $(l, v) \rightarrow_\sigma (l', v')$ if $(l, v) \in S$, $(l', v') \in S$, and we have that $(v, v') \in [\![\mathsf{Jump}(e)]\!]$;
  - **continuous**: for each nonnegative real $\delta \in \mathbb{R}^{\geq 0}$, we have $(l, v) \rightarrow_\delta (l', v')$ if $l = l'$, $(l, v) \in S$, $(l', v') \in S$, and there is a differentiable function $f : [0, \delta] \rightarrow \mathbb{R}^m$, with first derivative $\dot{f} : (0, \delta) \rightarrow \mathbb{R}^m$, such that the following conditions hold:
    - $f(0) = v$,
    - $f(\delta) = v'$,
    - for all reals $\epsilon \in (0, \delta)$, both $f(\epsilon) \in [\![\mathsf{Inv}(l)]\!]$ and $(f(\epsilon), \dot{f}(\epsilon)) \in [\![\mathsf{Flow}(l)]\!]$.

    The function f is called a *witness* for the transition $(l, v) \rightarrow_\delta (l', v')$.

In this transition system, we abstract continuous flows by transitions retaining only the information about the source, target and duration of each flow.

The paths contained in the timed transition system of a hybrid automaton H are formal representations of the possible trajectories of the hybrid system modeled by H, i.e., the evolutions of the state of the hybrid system along time. Formally, a *finite path*, noted λ, in the timed transition system $T = \langle S, S_0, \Sigma, \rightarrow \rangle$ is a finite sequence alternating between states and transition labels $s_0 \tau_0 s_1 \tau_1 \ldots \tau_{n-1} s_n$ such that at any i, $0 \leq i \leq n$, $s_i \in S$ and for any

i, $0 \leq i < n$, $(s_i, \tau_i, s_{i+1}) \in \rightarrow$. We call $n+1$ the *length of the path* and it is denoted by $|\lambda|$. This definition is extended to infinite paths as follows: an *infinite path* λ in the timed transition system T is an infinite sequence alternating between states and transition labels $s_0\tau_0 s_1\tau_1 \ldots \tau_{n-1}s_n \ldots$ such that for any $i \geq 0$: $s_i \in S$ and $(s_i, \tau_i, s_{i+1}) \in \rightarrow$. The length of an infinite path is $+\infty$. The *duration* of a (finite or infinite) path λ is the sum of time labels that appear along λ. That is, given $\lambda = s_0\tau_0 s_1\tau_1 \ldots s_n\tau_n \ldots$, let J be a subset of indices j in $\{0, 1, \ldots |\lambda|\}$ such that $\tau_j \in \mathbb{R}^{\geq 0}$, then the duration of λ is defined by $\mathsf{Duration}(\lambda) = \sum_{j \in J} \tau_j$. We say that a (finite or infinite) path λ is *initial* if its first state s_0 is an initial state of the TTS, i.e. $s_0 \in S_0$. We write $\mathsf{Path}_F(T)$ for the set of *finite initial paths* of S and $\mathsf{Path}_\infty(T)$ for the set of *infinite initial paths* of S.

Example 1. The following path belongs to $\mathsf{Path}_F([\![\mathsf{Tank}]\!])$:

$$(t_4, x \mapsto 20) \overset{(1)}{\underset{\mathsf{ON}}{\rightarrow}} (t_1, x \mapsto 20) \overset{(2)}{\underset{10}{\rightarrow}} (t_1, x \mapsto 88.59\ldots) \overset{(3)}{\underset{2.74\ldots}{\rightarrow}} (t_1, x \mapsto 100) \overset{(4)}{\underset{\mathsf{B}}{\rightarrow}}$$
$$(t_2, x \mapsto 100) \overset{(5)}{\underset{5}{\rightarrow}} (t_2, x \mapsto 100) \overset{(6)}{\underset{\mathsf{OFF}}{\rightarrow}} (t_3, x \mapsto 100) \overset{(7)}{\underset{8}{\rightarrow}} (t_3, x \mapsto 54.88\ldots)$$

Transition (1) is discrete: the control of the tank instantaneously changes from control location t_4 to control location t_1. The value of x remains equal to 20 due to the jump predicate $x' = x$ expressing that the value of x is left unchanged by the discrete jump. The witness function for time step (2) is $f(t) = 20e^{-0.075t} + 150(1 - e^{-0.075t})$ on the interval $[0, 10]$. For time step (3) the witness function is $f(t) = 88.59\ldots e^{-0.075t} + 150(1 - e^{-0.075t})$ on the interval $[0, 2.75]$. Transition (4) is a discrete change that is forced by the invariant $20 \leq x \leq 100$ that labels location t_1. The witness function for time step (5) is $f(t) = 100$ on the interval $[0, 5]$. Transition (6) is a discrete change that can occur at any time when in location t_2. The witness function for time step (7) is $f(t) = 100e^{-0.075t}$ on the interval $[0, 8]$.

Remark 1. If we are interested in the infinite behaviors of a hybrid automaton, then we are usually interested in infinite sequences of transitions that do not converge in time. In fact, trajectories during which an infinite number of discrete changes occur in a finite amount of time are not realistic. It is clear that if a controller takes discrete switches, say, at times $\frac{1}{2}, \frac{3}{4}, \frac{7}{8}, \frac{15}{16}, \ldots$, then it is not implementable. In this case, we say that the controller is Zeno. The *nonZenoness property* of an infinite path can be expressed as follows. Let $T\langle S, S_0, \Sigma, \rightarrow\rangle$ be a TTS and λ be an infinite path of T. The path λ is *nonZeno* if and only if $\mathsf{Duration}(\lambda) = +\infty$. The divergence of time is a *liveness* assumption [1], and it is the only liveness assumption we need to consider [25]. Algorithmic methods for checking nonzenoness properties of timed and hybrid automata are given in [36].

2.3 Composition

Nontrivial systems consist of several *interacting components* (three in our running example). We model each component as a hybrid automaton, and the components coordinate with each other by *shared variables* and *shared events*. The automaton for the thermometer and for the gas burner are given in Fig. 4. The thermometer uses the shared variable x to synchronize with the tank: the behavior of the thermometer depends on the evolution of the variable x whose evolution is governed by the hybrid automaton that models the tank. The flow of this variable is not constrained in the thermometer automaton. In our formalization, the thermometer samples the variable x once every $\frac{1}{10}$ time units and issues the event $\mathsf{DW93}$ if the temperature is below 93 degrees, issues the event $\mathsf{UP95}$ if the temperature is above 95 degrees, and issues an *internal* event ϵ in other cases (this event is not shared with other components). The sampling rate is enforced using the analog variable z that evolves with a derivative equal to 1. Such a variable counts time and is called a *clock*. The gas burner uses events to synchronize with the tank. The gas burner communicates with the tank by synchronizing control switches through the two shared events ON and OFF. The time needed for the gas burner to turn off or turn on is fixed at $\frac{1}{10}$ time units.

To formalize those intuitions, we use the notion of the product of two hybrid automata which is defined as follows.

Definition 4 [Automata-Product] Let $H^1 = \langle \mathsf{Loc}^1, \mathsf{Edge}^1, \Sigma^1, X^1, \mathsf{Init}^1, \mathsf{Inv}^1, \mathsf{Flow}^1, \mathsf{Jump}^1 \rangle$ and $H^2 = \langle \mathsf{Loc}^2, \mathsf{Edge}^2, \Sigma^2, X^2, \mathsf{Init}^2, \mathsf{Inv}^2, \mathsf{Flow}^2, \mathsf{Jump}^2 \rangle$ be two hybrid automata such that $\mathsf{Loc}^1 \cap \mathsf{Loc}^2 = \emptyset$. Their synchronized product, noted as $H^1 \otimes H^2$, is the hybrid automaton $H = \langle \mathsf{Loc}, \mathsf{Edge}, \Sigma, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump} \rangle$ defined as follows:

- $\mathsf{Loc} = \big\{ \{l^1, l^2\} \mid l^1 \in \mathsf{Loc}^1 \wedge l^2 \in \mathsf{Loc}^2 \big\}$.
- Edge is defined as follows: $(\{l_1^1, l_1^2\}, \sigma, \{l_2^1, l_2^2\}) \in \mathsf{Edge}$ iff either
  1. $\sigma \in \Sigma^1 \setminus \Sigma^2$, $(l_1^1, \sigma, l_2^1) \in \mathsf{Edge}^1$, and $l_1^2 = l_2^2$;
  2. $\sigma \in \Sigma^2 \setminus \Sigma^1$, $(l_1^2, \sigma, l_2^2) \in \mathsf{Edge}^2$, and $l_1^1 = l_2^1$;
  3. $\sigma \in \Sigma^1 \cap \Sigma^2$, $(l_1^1, \sigma, l_2^1) \in \mathsf{Edge}^1$ and $(l_1^2, \sigma, l_2^2) \in \mathsf{Edge}^2$.

  Conditions (1) and (2) express that unshared events (also called *internal events*) are interleaved while condition (3) expresses that shared events must occur simultaneously in the two automata.
- $\Sigma = \Sigma^1 \cup \Sigma^2$.
- $X = X^1 \cup X^2$.
- for any location $\{l^1, l^2\} \in \mathsf{Loc}$, we have that $\mathsf{Init}(\{l^1, l^2\}) = \mathsf{Init}^1(l^1) \wedge \mathsf{Init}^2(l^2)$.
- for any location $\{l^1, l^2\} \in \mathsf{Loc}$, we have that $\mathsf{Inv}(\{l^1, l^2\}) = \mathsf{Inv}^1(l^1) \wedge \mathsf{Inv}^2(l^2)$.
- for any location $\{l^1, l^2\} \in \mathsf{Loc}$, we have that $\mathsf{Flow}(\{l^1, l^2\}) = \mathsf{Flow}^1(l^1) \wedge \mathsf{Flow}^2(l^2)$.
- for any edge $(\{l_1^1, l_1^2\}, \sigma, \{l_2^1, l_2^2\}) \in \mathsf{Edge}$, we have that:

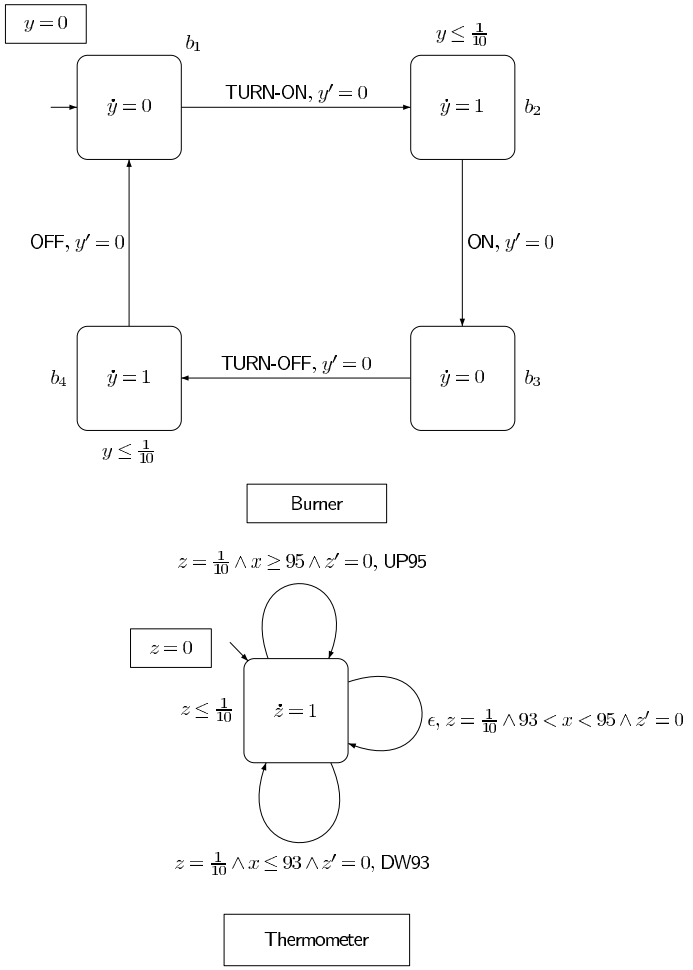


Fig. 4. Hybrid automata for the burner and the thermometer

1. $\mathsf{Jump}(\{l_1^1, l_1^2\}, \sigma, \{l_2^1, l_2^2\}) = \mathsf{Jump}((l_1^1, \sigma, l_2^1)) \wedge \bigwedge_{x \in X^2 \setminus X^1} x' = x$ if $\sigma \in \Sigma^1 \setminus \Sigma^2$;
2. $\mathsf{Jump}(\{l_1^1, l_1^2\}, \sigma, \{l_2^1, l_2^2\}) = \mathsf{Jump}((l_1^2, \sigma, l_2^2)) \wedge \bigwedge_{x \in X^1 \setminus X^2} x' = x$ if $\sigma \in \Sigma^2 \setminus \Sigma^1$;
3. $\mathsf{Jump}(\{l_1^1, l_1^2\}, \sigma, \{l_2^1, l_2^2\}) = \mathsf{Jump}((l_1^1, \sigma, l_2^1)) \wedge \mathsf{Jump}((l_1^2, \sigma, l_2^2))$ if $\sigma \in \Sigma^1 \cap \Sigma^2$;

Conditions 1 and 2 express that discrete changes that are local to one automaton have the enabling condition and the effect described by the jump predicate of that automaton and the variables which are not shared remain unchanged. Condition 3 expresses that discrete changes shared

by the two automata have as enabling condition the conjunction of the enabling conditions of each discrete change. Their effect is the conjunction of the effects of each discrete change.

In our example, we obtain the complete system by composing the three automata. It is easy to show that the product operation that we have defined is commutative and associative, so we can write $\mathsf{Sys} = \mathsf{Tank} \otimes \mathsf{Burner} \otimes \mathsf{Thermo}$. Fig. 5 shows the hybrid automaton obtained by composing the automaton for the tank and the automaton for the thermometer. We have omitted transitions that are incompatible with the invariant of their starting location. That is, edges $e = (l, \sigma, l')$ such that $[\![\mathsf{Jump}(e) \wedge \mathsf{Inv}(l)]\!] = \emptyset$ are not depicted.

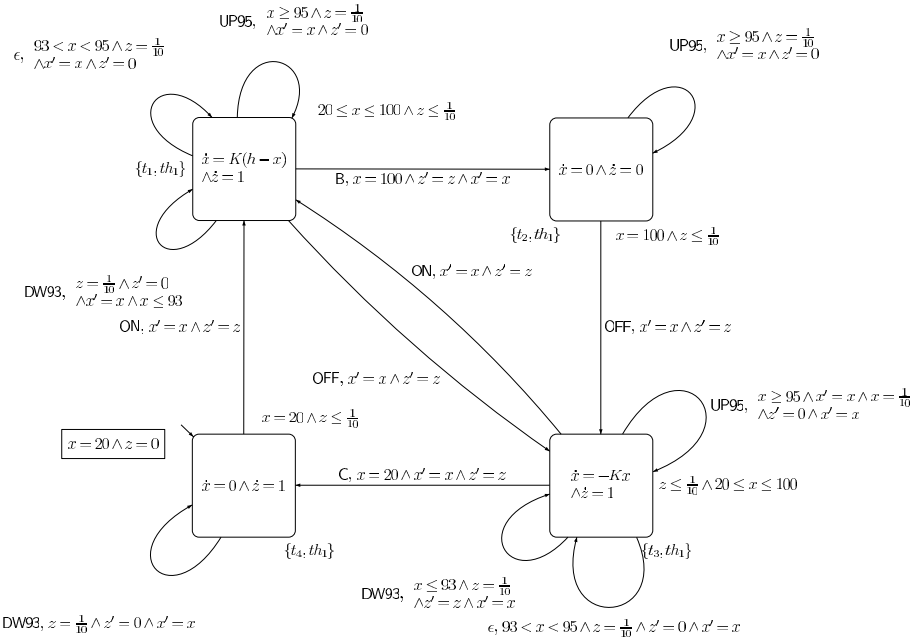


Fig. 5. Product of tank and thermometer

3 Properties of Hybrid Systems

Properties assign values to trajectories of hybrid systems. In this introduction, we restrict ourselves to properties that classify trajectories as *good* or *bad* according to whether or not they stay in a given set of (good) states. Those properties are called *safety properties* [1], and they are the most important class of properties when considering *safety critical systems*.

Let us go back to our running example. Now that we have a complete model of our system, we would like to design a controller that enforces some desired behaviors. The controller will be an additional hybrid automaton that, when composed with the automata modeling our system, must enforce the following properties on the trajectories of the entire system:

- ($R1$) the temperature in the tank must never reach 100 degrees;
- ($R2$) after 15 seconds of operation, the system must be in *stable regime*, which means that the temperature of the water in the tank must always stay between 91 and 97 degrees;
- ($R3$) during this stable regime, the burner is never continuously ON for more than two seconds.

The three properties above are *safety properties*. They impose that the system should stay within a set of *safe states*, or equivalently, that the system should never enter a set of *bad states* (states where the safety property is violated). This is clear for property ($R1$) where the bad states are the states where the value of x is greater than or equal to 100. We will see later that this is also the case for the other two properties. In this chapter, we only focus on safety properties. Pointers to the literature are given in the last section for other classes of properties.

We propose in Fig. 6, a possible controller for our system. The behavior of this controller is as follows. The controller observes two events coming from the thermometer (UP95, DW93) and issues two events toward the gas burner (TURN-ON, TURN-OFF). Initially, the controller waits in location c_1 until it sees the event DW93. When this event occurs, the control switches instantaneously to location c_2. There, it immediately switches to c_3 by emitting the event TURN-ON toward the gas burner. In location c_3, the controller ignores the event DW93 and waits for the event UP95. When this event takes place, the control moves to location c_4 where it instantaneously emits the event TURN-OFF toward the gas burner.

In the next section, we show how to formalize the requirements expressed informally above and how to prove, using algorithmic methods, that the controller we propose fulfills those requirements.

3.1 Safety properties and monitors

Safety properties

To formalize safety properties, we need some more notation. Let $T = \langle S, S_0, \Sigma, \rightarrow \rangle$ be a TTS. Let $\lambda = s_0 \tau_0 s_1 \tau_1 \dots s_n$ be a finite path in T. We denote $\mathsf{State}(\lambda)$ for the set of states that *appear* along the path λ. We say that a path λ *reaches* a state s if $s \in \mathsf{State}(\lambda)$. We say that a state s is *reachable* in T if $s \in \bigcup_{\lambda \in \mathsf{Path}_F(T)} \mathsf{State}(\lambda)$. The set of states that are reachable in T is noted $\mathsf{Reach}(T)$. A set of states $R \subseteq S$ is called a *region*. We note $\overline{R}$ for the

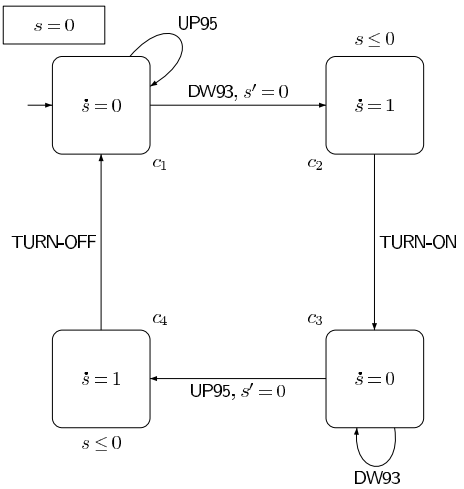


Fig. 6. A controller for the system

complement of R in the state space of T, that is, $\overline{R} = S \setminus R$. We say that T is *safe for* R iff $\mathsf{Reach}(T) \subseteq R$. A region R is *reachable* in T iff $R \cap \mathsf{Reach}(T) \neq \emptyset$.

Definition 5 [Verification Problems] Let H be a hybrid automaton with TTS $[\![H]\!]$ whose state space is S, and let $R \subseteq S$ be a region. The *safety problem* associated to R asks whether $\mathsf{Reach}([\![H]\!]) \subseteq R$. The *reachability problem* associated to R asks whether $\mathsf{Reach}([\![H]\!]) \cap R \neq \emptyset$.

Those two problems are *dual* in the following formal sense.

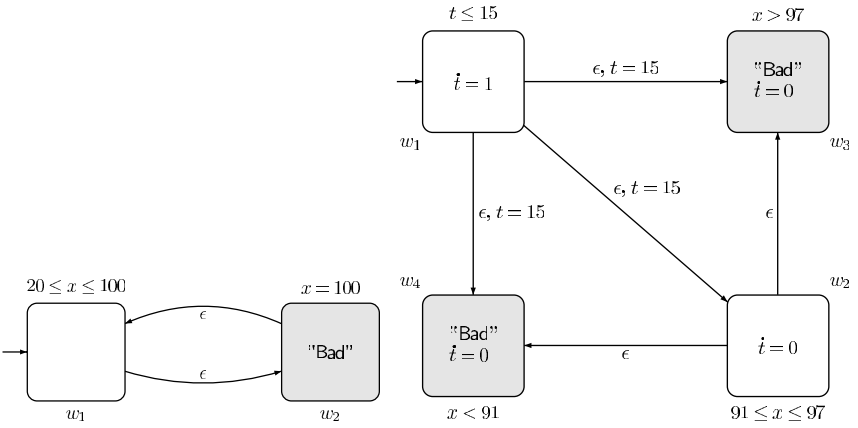
Theorem 1. *For any* TTS T*, for any region* R *of* T*,* $\mathsf{Reach}(T) \subseteq R$ *iff* $\mathsf{Reach}(T) \cap \overline{R} = \emptyset$.

Hence, solving a safety problem boils down to solving its dual reachability problem. In that reachability problem, the region $\overline{R}$ is often called the set of *bad states*.

Monitors

In order to formalize safety requirements, it is often very convenient to use a *monitor automaton*, also often called an *observer*, that "watches" the trajectories of the system and enters "Bad" locations whenever one trajectory violates a given safety property. Safety verification is then reduced to deciding the reachability of a set of "Bad" locations.

In Fig. 7(a), 7(b), and 7(c), we give the monitors for the safety requirements $(R1)$, $(R2)$, and $(R3)$ respectively. The automaton Moni_1 monitors the value of variable x whose dynamics is defined in the tank automaton. As soon as x reaches the value 100, the control of the monitor can move to location



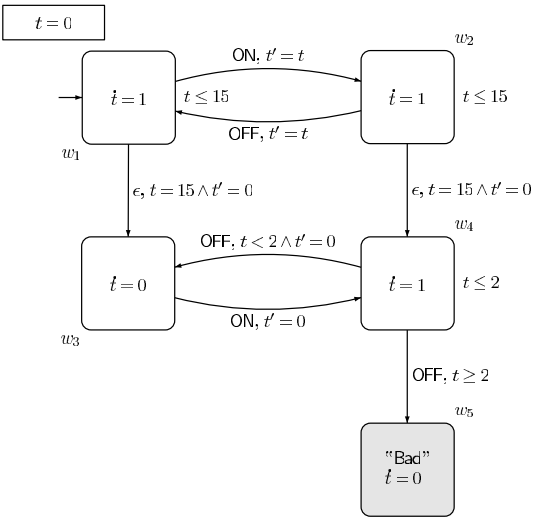


Fig. 7. Monitors for the safety properties $(R1)$, $(R2)$, and $(R3)$

w_2 which is a Bad location. Thus to verify property $(R1)$, we have to establish that no state in which the control of Moni_1 is in location w_2 is reachable in $[\![\mathsf{Tank} \otimes \mathsf{Burner} \otimes \mathsf{Thermo} \otimes \mathsf{Controller} \otimes \mathsf{Moni}_1]\!]$. In that case, we know that the controller ensures requirement $(R1)$. The automaton Moni_2 initially maintains a variable t that counts the time elapsed since the initialization of the system. When this variable reaches value 15 (the system was started 15 seconds ago), the control has to leave location w_1. If the value of variable x (the temperature of the water inside the tank) at that time is between 91 and 97, the control moves to location w_2 and the control can stay there only if the

temperature stays within this interval of values. On the other hand, if the value is, or becomes, less than 91, it moves to location w_3, and if the value is, or becomes, greater than 97, it moves to location w_4. Locations w_3 and w_4 are the Bad locations. It is clear that if no state in which the control of Moni_2 is either w_3 or w_4 is reachable in $[\![\mathsf{Tank} \otimes \mathsf{Burner} \otimes \mathsf{Thermo} \otimes \mathsf{Controller} \otimes \mathsf{Moni}_2]\!]$, then we know that the controller ensures requirement $(R2)$. Finally, the automaton Moni_3 works as follows. For the first 15 seconds, the control stays in location w_1, if the burner is OFF, or in location w_2, if the burner is ON. After 15 seconds, the control moves to location w_3 or w_4. In w_3, each time the event ON occurs, the variable t is reset and the control moves to location w_4 where t counts time. There, the monitor waits for the next OFF event. If this next OFF event occurs within 2 time units ($t < 2$), then the control moves back to location w_3 where the monitor waits for the next ON event. On the other hand, if the event OFF occurs after 2 time units ($t \geq 2$), then the control of the monitor moves to location w_5, a "Bad" location. Again, it is clear that our system satisfies requirement $(R3)$ if no state where the control of Moni_3 is in location w_5 can be reached in $[\![\mathsf{Tank} \otimes \mathsf{Burner} \otimes \mathsf{Thermo} \otimes \mathsf{Controller} \otimes \mathsf{Moni}_3]\!]$.

3.2 How do we solve reachability problems?

We have seen that safety verification problems can be reduced to reachability problems. We introduce here some basic notions useful for reachability problems. Given a TTS $T = \langle S, S_0, \Sigma, \rightarrow \rangle$, we define the following two operators:

- the *direct successor operator* $\mathsf{Post}_T : 2^S \rightarrow 2^S$, is an operator that, given a set of states, returns the set of direct successors of those states in T. Formally, for any $S' \subseteq S$, we have that

  $$\mathsf{Post}_T(S') = \big\{ s \in S \mid \exists s' \in S' : (\exists \sigma \in \Sigma : s' \rightarrow_\sigma s \vee \exists \delta \in \mathbb{R}^{\geq 0} : s' \rightarrow_\delta s) \big\}.$$

- the *direct predecessor operator* $\mathrm{Pre}_T : 2^S \rightarrow 2^S$, is an operator that, given a set of states, returns the set of direct predecessors of those states in T. Formally, for any $S' \subseteq S$, we have that

  $$\mathrm{Pre}_T(S') = \big\{ s \in S \mid \exists s' \in S' : (\exists \sigma \in \Sigma : s \rightarrow_\sigma s' \vee \exists \delta \in \mathbb{R}^{\geq 0} : s \rightarrow_\delta s') \big\}.$$

The set of reachable states of a hybrid automaton H can be described as the least solution (for the subset order over sets of states) of equations constructed using the direct successor or predecessor operators:

- The set of reachable states of a hybrid automaton H with TTS $[\![H]\!] = \langle S, S_0, \Sigma, \rightarrow \rangle$ can be described as the least solution of the following equation:

  $$X = (S_0 \cup \mathsf{Post}_{[\![H]\!]}(X)), \tag{1}$$

  where X ranges over sets of states.

- Symmetrically, the set of states of this automaton that can reach a given region R can be described as the least solution of the following equation:

$$X = (R \cup \mathrm{Pre}_{[\![H]\!]}(X)), \tag{2}$$

where X ranges over sets of states.

As the direct successor and the direct predecessor operators are monotone for the subset order, we know by the Tarsky theorems that the least solutions of those equations can be obtained by successive approximations. Unfortunately, this does not mean that we can effectively solve those equations. In fact, the fixpoint is not necessarily reached within a finite number of steps. In general, reachability problems are undecidable for even the simplest class of hybrid automata (we give detailed references to the literature later). Even applying the direct successor or predecessor operator to a region one time may be very difficult as it amounts to solving general differential equations. We do not know how to do that in general. This is why subclasses of hybrid systems for which we know how to compute direct successors or predecessors of regions have been studied in the literature [4]. In the next section, we study a particularly interesting one, the *rectangular hybrid automata* [34, 45].

4 Rectangular Hybrid Automata

4.1 Syntactic restrictions

An interval is a convex non-empty subset of the positive real numbers with greatest lower bound in $\mathbb{Q} \cup \{-\infty\}$ and least upper bound in $\mathbb{Q} \cup \{+\infty\}$. As usual, intervals can be denoted by $(a, b), [a, b), (a, b]$ or $[a, b]$ where $a \in \mathbb{Q} \cup \{-\infty\}$ and $b \in \mathbb{Q} \cup \{+\infty\}$, and $a \leq b$. Let I be an interval, we note $glb(I)$ for the *greatest lower bound* of I and $lub(I)$ for the *least upper bound* of I. Let X be a set of variables, we note $\mathsf{Rect}(X)$ for the following set of formulas:

$$\mathsf{Rect}(X) \ni \phi_1, \phi_2 := \bot \mid \top \mid x \in I \mid \phi_1 \wedge \phi_2,$$

where x belongs to the set of variables X, and I is an interval. Those formulas are called *rectangular predicates*. The set of formula $\mathsf{Rect}(\dot{X})$ is defined in the same way, replacing X by $\dot{X}$. Those formulas are called *rectangular flow predicates*. We need a last set of formulas. We denote by $\mathsf{UpdateRect}(X)$, the following set of formulas:

$$\mathsf{UpdateRect}(X) \ni \phi_1, \phi_2 := \bot \mid \top \mid x \in I \mid x' \in I \mid x' = x \mid \phi_1 \wedge \phi_2,$$

where x belongs to the set of variables X, x' belongs to X' the set of primed copies of variables in X, and I is an interval. Formulas from this set are called *rectangular update predicates*.

Definition 6 [Rectangular Automaton] A rectangular automaton is a hybrid automaton $H = \langle \mathsf{Loc}, \mathsf{Edge}, \Sigma, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump} \rangle$ where for any $l \in \mathsf{Loc}$, $\mathsf{Init}(l)$ and $\mathsf{Inv}(l)$ are rectangular predicates over X, that is, formulas taken in $\mathsf{Rect}(X)$, for any edge $e \in \mathsf{Edge}$, $\mathsf{Jump}(e)$ is a rectangular update predicate over X, that is a formula taken in $\mathsf{UpdateRect}(X)$, and finally, for any location $l \in \mathsf{Loc}$, $\mathsf{Flow}(l)$ is a rectangular flow predicate over $\dot{X}$, that is, a formula taken in $\mathsf{Rect}(\dot{X})$.

It is easy to show that the composition of two rectangular automata is again a rectangular automaton. The hybrid automata for the gas burner, the thermometer, the controller, and the three monitors are all rectangular hybrid automata.

4.2 Reachability analysis of rectangular hybrid automata

The computation of the Pre and Post operators is easier in the case of rectangular hybrid automata. For that class of hybrid automata, we are able to define a semi-algorithm (no guarantee of termination) for reachability. This semi-algorithm manipulates regions that are infinite sets of states. Therefore, we need a way to represent regions in a symbolic way.

A *linear term* over the set of variables X is a linear combination of the variables in X with integer coefficients. A *linear formula* over X is a boolean combination of inequalities between linear terms over X. Given a linear formula Ψ, we write $[\![\Psi]\!]$ for the set of valuations v of the variables in X such that $v \models \Psi$. If we allow quantifiers with linear formulas, we obtain the *theory of reals with addition*, noted $\mathsf{T}(\mathbb{R}, 0, 1, +, \leq)$. Note that rectangular predicates, rectangular flow predicates, and rectangular update predicates are linear formulas over X, $\dot{X}$, and $X \cup X'$, respectively.

Let $H = \langle \mathsf{Loc}, \mathsf{Edge}, \Sigma, X, \mathsf{Init}, \mathsf{Inv}, \mathsf{Flow}, \mathsf{Jump} \rangle$ be a rectangular automaton. A *symbolic region* $\mathcal{R}$ of H is a finite set $\{(l, \Psi_l) \mid l \in \mathsf{Loc}\}$ of pairs, where $l \in \mathsf{Loc}$ is a location of the automaton and Ψ_l is a linear formula such that $[\![\Psi_l]\!] \subseteq [\![\mathsf{Inv}(l)]\!]$. Let $l \in \mathsf{Loc}$ and let $\mathsf{Flow}(l)$ be the rectangular flow predicate that labels l. We denote by $[\![\mathsf{Flow}(l)]\!]\ (x)$ the set of values $\{\dot{v}(x) \mid \dot{v} \in [\![\mathsf{Flow}(l)]\!]\}$, that is the set of possible values of the first derivative of variable x when the control is in location l. It is easy to show that this set is an interval of the real numbers with rational lower and upper bounds.

Given a location $l \in \mathsf{Loc}$ and a set of valuations $V \subseteq [X \to \mathbb{R}]$, such that $V \subseteq [\![\mathsf{Inv}(l)]\!]$: the *forward time closure*, noted $\langle V \rangle_l^{\nearrow}$ of V at l is the set of valuations of variables in X that are reachable from some valuation $v \in V$ by letting time pass:

$$\left\{ v' \mid \exists v \in V, t \in \mathbb{R}^{\geq 0} : \forall x \in X : \begin{array}{l} \langle V \rangle_l^{\nearrow} = \\ \wedge v(x) + t \times glb([\![\mathsf{Flow}(l)]\!]\ (x)) \prec_x^1 v'(x) \\ \wedge v'(x) \prec_x^2 v(x) + t \times lub([\![\mathsf{Flow}(l)]\!]\ (x)) \\ \wedge v' \in [\![\mathsf{Inv}(l)]\!] \end{array} \right\}$$

where
$$\prec_x^1 = \begin{cases} \leq \text{ if } glb(\llbracket \mathsf{Flow}(l) \rrbracket\ (x)) \in \llbracket \mathsf{Flow}(l) \rrbracket\ (x), \text{ i.e., the interval is left closed} \\ < \text{ if } glb(\llbracket \mathsf{Flow}(l) \rrbracket\ (x)) \notin \llbracket \mathsf{Flow}(l) \rrbracket\ (x), \text{ i.e., the interval is left open} \end{cases}$$
where
$$\prec_x^2 = \begin{cases} \leq \text{ if } lub(\llbracket \mathsf{Flow}(l) \rrbracket\ (x)) \in \llbracket \mathsf{Flow}(l) \rrbracket\ (x), \text{ i.e., the interval is right closed} \\ < \text{ if } lub(\llbracket \mathsf{Flow}(l) \rrbracket\ (x)) \notin \llbracket \mathsf{Flow}(l) \rrbracket\ (x), \text{ i.e., the interval is right open} \end{cases}$$

The set above can be defined inside $\mathsf{T}(\mathbb{R}, 0, 1, +, \leq)$. As $\mathsf{T}(\mathbb{R}, 0, 1, +, \leq)$ admits quantifier elimination, it is clear that given any linear formula Ψ, we can construct a linear formula Φ such that $\llbracket \Phi \rrbracket = \langle \llbracket \Psi \rrbracket \rangle_l^{\nearrow}$.

Given an edge $e \in \mathsf{Edge}$ and a set of valuations $V \subseteq [X \to \mathbb{R}]$, the *postcondition* $\mathsf{post}_e(V)$ of V with respect to e is the set of valuations that are reachable from some valuation $v \in V$ by taking the discrete transition e:

$$\mathsf{post}_e(V) = \{v' \mid \exists v \in V : (v, v') \in \llbracket \mathsf{Jump}(e) \rrbracket\}.$$

Again, as $\mathsf{T}(\mathbb{R}, 0, 1, +, \leq)$ admits quantifier elimination, and for any edge e, $\mathsf{Jump}(e)$ is a rectangular update predicate over X, and so a linear formula over $X \cup X'$, it is clear that if we are given a linear formula Ψ, then we can construct a linear formula Φ such that $\llbracket \Phi \rrbracket = \mathsf{post}_e(\llbracket \Psi \rrbracket)$.

We can now define the forward time closure and the edge postcondition operators of H over symbolic regions. Let $\mathcal{R} = \{(l, \Psi_l \wedge \mathsf{Inv}(l)) \mid l \in \mathsf{Loc}\}$ be a symbolic region of H:

- $\langle \mathcal{R} \rangle^{\nearrow} = \bigcup_{l \in \mathsf{Loc}} \{(l, \langle \llbracket \Psi_l \rrbracket \rangle_l^{\nearrow})\}$
- $\mathsf{post}(\mathcal{R}) = \bigcup_{e=(l,\sigma,l') \in \mathsf{Edge}} \{(l', \mathsf{post}_e(\Psi_l))\}$

From those two operators, we can define our symbolic post operator for rectangular automata as follows. Let $\mathcal{R} = \{(l, \Psi_l) \mid l \in \mathsf{Loc}\}$ be a symbolic region of H:

$$\mathsf{Post}(\mathcal{R}) = \mathsf{post}(\langle \mathcal{R} \rangle^{\nearrow}).$$

Now, we can use the Tarsky fixpoint theorem to find the least solution of equation (1) by successive approximations defined as follows:

- $\mathcal{R}_0 = \{(l, \mathsf{Init}(l)) \mid l \in \mathsf{Loc}\}$
- for any integer $i > 0$, $\mathcal{R}_i = \mathcal{R}_{i-1} \cup \mathsf{Post}(\mathcal{R}_{i-1})$

This approximation schema defines naturally a semi-algorithm for reachability. This algorithm is given in Fig. 8.

4.3 Rectangular hybrid automata as abstractions

Let us go back to our running example. Remember that the automata for the burner, the thermometer, the controller, and the three monitors that we have defined above are all in the class of rectangular hybrid automata. The only automaton of our example which is outside the class of rectangular hybrid automata is the automaton for the tank.

A symbolic algorithm for reachability
begin
$R := \{(l, \mathsf{Init}(l) \wedge \mathsf{Inv}(l)) \mid l \in \mathsf{Loc}\}$;
$Prec := \emptyset$;
while $[\![R]\!] \not\subseteq [\![Prec]\!]$ **do**
$Prec := Prec \cup R$;
$R := Post(R)$;
od
if $Bad \cap Prec = \emptyset$ **then** *return*(OK) **else** *return*(KO); **fi**
where $[\![R]\!] \not\subseteq [\![Prec]\!]$ *holds if there exist* $(l, \Psi) \in R$ *and* $(l, \Psi') \in Prec$ *such that* $\forall x_1, \ldots, x_m : \Psi(x_1, \ldots, x_n) \rightarrow \Psi'(x_1, \ldots, x_n)$ *is not a valid formula.*

Fig. 8. Semi-algorithm for the reachability analysis of rectangular hybrid automata

In this subsection, we show how to *approximate* complex dynamics with rectangular dynamics in a systematic way. Those systematic approximations allow us to use automatic tools, like HyTech [29], to analyze approximated systems and, in a lot of practical cases, to infer the important properties of the original (complex) systems. This methodology is closely related to the theory of abstract interpretation studied by computer scientists [20] and the approximation techniques used in analysis of dynamical systems [40].

We introduced here an approximation schema known as the *rectangular phase-portrait approximation* scheme; see [30] for more details. The idea of this approximation scheme can be stated as follows. For each control mode of the hybrid automaton that we want to approximate, the state space is partitioned into rectangular regions, and within each region, the flow field is over-approximated using rectangular flows. Those approximations may be obtained manually, using techniques from dynamical system theory, or in some cases automatically, when lower and upper bounds on derivatives can be obtained from bounds on the value of variables within rectangular regions. The approximations can be arbitrarily precise by approximating over suitably small regions of the state space.

Let us illustrate that approximation schema on our running example. Let us consider the location t_1 of the tank. In this location, we know that the possible values for x, the temperature of the water within the tank, are such that $20 \leq x \leq 100$ (this is given by the invariant that labels the location) and the flow of x is given by the flow predicate $\dot{x} = K(h - x)$. As the second derivative of x in the interval $[20, 100]$ is never zero, we know that the minimal value of the first derivative of x in this interval occurs when $x = 100$ and the maximal derivative occurs when $x = 20$. Remember that we have fixed the value of the constant K to 0.075 and h to 150. With those constants, we know that the values of the first derivative of x within $[20, 100]$ are bounded from below by 3.75 and from above by 9.75. It means that if we replace the flow predicate of location t_2 by $\dot{x} \in [\frac{375}{100}, \frac{975}{100}]$, or by $\dot{x} \in [3, 10]$ to keep things simple, then we are sure that the resulting automaton will define at

least the trajectories defined by the original automaton. We can repeat this schema for each location of the original automaton. In this way we obtain a rectangular hybrid automaton that overapproximates the behavior of our original model in the sense that any trajectory of the original automaton can be mimicked by the approximating automaton (and so is a trajectory of the approximating automaton). In this introduction the notion of approximations is left informal; it can be formalized using notions like simulations [43], and we refer the interested reader to [30] for a correctness proof. The automaton obtained by this schema is given in Fig. 9 and is noted RectTank.

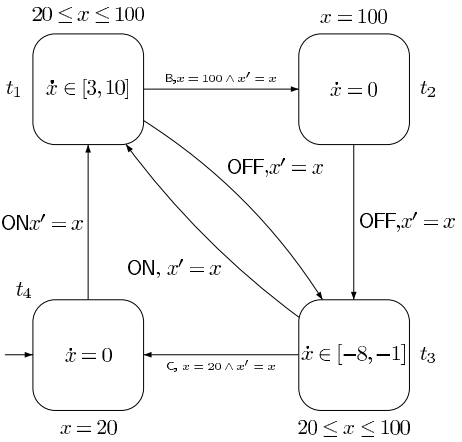


Fig. 9. Rectangular automaton RectTank for the tank

Let us now analyze the behaviors of our system approximated as a product of rectangular hybrid automata. This model can be analyzed using the tool HyTech [29]. HyTech is a model-checking tool for the reachability analysis of linear hybrid automata, a class of hybrid automata that subsumes the class of rectangular hybrid automata. HyTech allows us to describe each component of the system directly as a rectangular automaton in a textual syntax and to formalize reachability questions using a simple (and yet powerful) script language.

For our analysis of the tank system, we consider the product of each of the three monitors Moni_i, $1 \leq i \leq 3$, of Fig. 7(c), with the system $\mathsf{RectTank} \otimes \mathsf{Burner} \otimes \mathsf{Thermo} \otimes \mathsf{Controller}$. Again, it is easy to show that since RectTank overapproximates the behaviors of Tank, and if "Bad" locations are not reachable in $\mathsf{RectTank} \otimes \mathsf{Burner} \otimes \mathsf{Thermo} \otimes \mathsf{Controller} \otimes \mathsf{Moni}_i$ then "Bad" is also not reachable in $\mathsf{Tank} \otimes \mathsf{Burner} \otimes \mathsf{Thermo} \otimes \mathsf{Controller} \otimes \mathsf{Moni}_i$. This means that if we can prove that a safety requirement is verified in the approximated system, then it is also verified for the original system.

When running the three verification tasks in HyTech, only the verification task of property $(R1)$ is positive in the approximated system; the two other properties turn out to be false in this approximation. HyTech provides witness trajectories that lead to bad states, that is, trajectories where the control of monitors Moni_2 and Moni_3 enter bad locations. If we look carefully at those trajectories, we can see that they are not possible in the original system. In particular, there are continuous transitions that cannot be mimicked by the concrete system. Those paths are present because of the overapproximation. To rule out those *spurious* paths, we have to refine our initial approximation and get closer to the real dynamics of the temperature of the water in the tank. For that purpose, we proceed as follows. As suggested above, we must partition the state space in smaller rectangular regions to capture more precisely the first derivative of x. To do that, we need to split some control modes of our original automaton. Consider the control mode modeled by location t_1, that is, when the burner is ON and the temperature is rising following the flow predicate $\dot{x} = K(h - x)$. Instead of considering only the rectangular region $20 \leq x \leq 100$, we will consider the four regions $20 \leq x \leq 50$, $50 \leq x \leq 91$, $91 \leq x \leq 95$, and finally $95 \leq x \leq 100$. For those regions, we can approximate, using the same reasoning as above, the first derivative of x by the following rectangular flow predicates: $\dot{x} \in [7, 10]$ for the first region, $\dot{x} \in [4, 8]$ for the second region, $\dot{x} \in [4, 5]$ for the third region, and $\dot{x} \in [3, 5]$ for the last region. This splitting is depicted in Fig. 10. Internal actions are taken to move the control from one region to the next when the boundaries of the region are reached. We can also apply this process to location t_3 and split this control mode into 3 locations as follows. Instead of the region $20 \leq x \leq 100$, we use the regions $20 \leq x \leq 91$, $91 \leq x \leq 97$, and finally $97 \leq x \leq 100$. The flow predicates that we obtain are, respectively, $\dot{x} \in [-7, -1]$, $\dot{x} \in [-8, -6]$, and $\dot{x} \in [-8, -7]$. Finally, we obtain a new overapproximating automaton that we denote $\mathsf{RectTank}_2$. Fig. 11 shows how the dynamics of the temperature of the water is approximated within the refined rectangular automaton for the tank within location t_1.

Now if we test the reachability of the Bad location of the monitors Moni_i, $1 \leq i \leq 3$, in $\mathsf{Tank} \otimes \mathsf{Burner} \otimes \mathsf{Thermo} \otimes \mathsf{Controller} \otimes \mathsf{Moni}_i$, with HyTech, we obtain that the "Bad" locations are not reachable in the three cases. This allows us to conclude that our controller is correct for the original (complex) system.

5 Beyond This Introduction

We close this chapter by referencing the literature. The interested reader will find in this section references to articles that will allow her/him to go beyond this introduction. We have organized the section into subsections devoted to active areas of research in the field of hybrid automata. Some references below have already been given above. These references are not intended to be

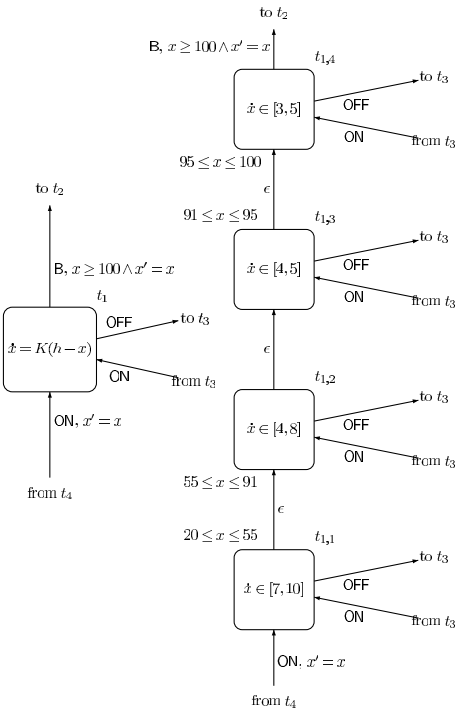


Fig. 10. Refinement by location splitting

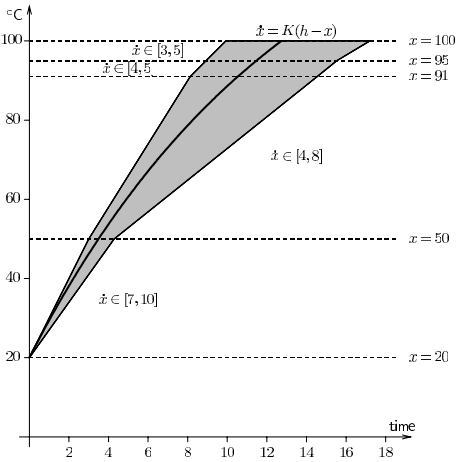


Fig. 11. Approximation of the dynamics by rectangles with rectangular regions

exhaustive (some important works may have been forgotten), but they are, from the point of view of the author, natural papers to look at in order to delve deeper into notions only sketched in this introductory chapter.

5.1 Analysis: Subclasses, decidability and complexity results

In [6, 7], Alur and Dill have introduced timed automata. This was the first proposal to extend finite state automata with continuous variables. Timed automata are a subclass of hybrid automata where continuous variables are *clocks*, that is, continuous variables that have constant slopes equal to 1 (they count time), values of clocks are compared to constants, and the only updates allowed are resets to 0. The reachability problem for timed automata is decidable (it is PSPACE-complete). Symbolic procedures to analyze timed automata are given in [36]. The first proposition to extend timed models to more general hybrid models can be found in [42]. Rectangular hybrid automata have been proposed in [45]. The reachability problem of rectangular hybrid automata is undecidable in the general case, but it is decidable for the subclass of initialized rectangular hybrid automata [34]. Other interesting subclasses of hybrid automata that can be analyzed algorithmically are integration graphs [41] and dynamical systems with piecewise constant derivatives [13]. More details and pointers about analysis and decidability results related to subclasses of hybrid automata can be found in [4, 28].

5.2 Beyond monitors: Temporal logics and real-time logics

Temporal logics have proven useful for specification and verification of reactive systems [19, 44]. In this introduction, we have focused only on the verification of the important class of safety properties: many more involved properties reduce to safety properties if progress of time is ensured [25]. Nevertheless, there has been a lot of research on suitable formalisms to express properties of hybrid systems. In particular, temporal logics have been extended for real-time. The reader interested in real-time logics is referred to [3, 8, 27, 38, 46, 47] for definitions and verification methods related to those logics. As an illustration of the expressive power of real-time logics, we give here the formalization of the three requirements of our running example in the logic MITL [8]. The following formulas are requirements that any infinite trajectory of the tank system must verify. The $\Box$ operator is read as "Always" (in the future), $\Box_{\geq 15}$ is read as "always after 15 time units", $\Diamond_{<2}$ is read as "there exists a state distant of less than 2 time units". The three requirements are then formalized as follows:

- $\Box(x < 100)$, meaning that in any trajectory, in any state, the temperature of the water is strictly less than 100 degrees;
- $\Box_{\geq 15}(91 \leq x \leq 97)$, meaning that in any trajectory, after 15 time units, the temperature of the water is always between 91 and 97 degrees;

- $\Box_{\geq 15}(\mathsf{ON} \rightarrow \Diamond_{<2}\mathsf{OFF})$, meaning that, in any trajectory, after 15 time units, any state where the burner is ON is followed within 2 time units by a state where the burner is OFF.

5.3 Equivalence relations and abstraction

Abstraction methods are used to simplify models and make their analysis more tractable. Several equivalence relations have been studied for subclasses of hybrid systems. For example, it can be shown that transition systems of timed automata admit finite state abstractions, called region graphs, that are time-abstract bisimilar, see [7] for details. Those equivalence results are used to prove decidability of verification problems on subclasses of hybrid automata [26,32] and allow the use of well-known model-checking procedures that are guaranteed to terminate in the presence of finite quotients [35]. Other techniques that are not exact but use overapproximations have been proposed and have proven useful in practice: the approximation schema proposed in Section 4.3 is detailed and proven correct in [30]. Other interesting works about overapproximations can be found, among others, in [5, 9, 24].

5.4 Control synthesis

In this introduction, we have shown how we can model and verify controllers using hybrid automata. In our example, we have proposed a controller for the system and proven that the controller was correct for a list of requirements. A more ambitious goal than algorithmic (controller) verification is algorithmic (controller) synthesis. References about control synthesis include [14, 17, 31, 33, 50].

5.5 Semantics and robustness

The semantics of hybrid automata that we defined in this chapter can be described as *perfect*. For example, it is possible to model, with this semantics, a controller that takes a given transition when a variable of the environment has exactly a given value. This can be considered as unrealistic because any implementation of such a controller will measure its environment through sensors that have *finite precision*. Alternative semantics that can be considered as *robust* are proposed in [11, 22, 23, 37].

5.6 Tool support and case studies

Several tools for the automatic analysis of hybrid automata have been implemented. The tools KRONOS [21] and UPPAAL [16] can be used to analyze the subclass of timed automata. The tool HYTECH [29] allows the analysis of linear hybrid automata. The tool CHARON [10] and the tool **d/dt** [12] allow the analysis of a more general class of hybrid automata.

Those tools have been applied successfully to a large set of case studies in a variety of application domains. Interesting case studies can be found in [2, 15, 39, 48, 49].

Acknowledgments

I would like to thank Alessandro Cimatti for reading a first version of this chapter and for giving invaluable advice to improve it. I would also like to thank Bram De Wachter, Laurent Doyen, Pierre Ganty, and Gilles Geeraerts for carefully reading a draft of this chapter and for helping me with several of the figures.

References

1. B. Alpern and F. B. Schneider. Defining liveness. *Information Processing Letters*, 21:181–185, 1985.
2. R. Alur, C. Belta, F. Ivančić, V. Kumar, M. Mintz, G. J. Pappas, H. Rubin, and J. Schug. Hybrid modeling and simulation of biomolecular networks. *Lecture Notes in Computer Science*, 2034:19–31, 2001.
3. R. Alur, C. Courcoubetis, and D. L. Dill. Model checking in dense real time. *Information and Computation*, 104(1):2–34, 1993.
4. R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138:3–34, 1995.
5. R. Alur, T. Dang, and F. Ivančič. Counter-example guided predicate abstraction of hybrid systems. In *TACAS: International Workshop on Tools and Algorithms for the Construction and Analysis of Systems, LNCS*, 2003.
6. R. Alur and D. L. Dill. Automata for modeling real-time systems. In M.S. Paterson, editor, *ICALP 90: Automata, Languages, and Programming*, Lecture Notes in Computer Science 443, pages 322–335. Springer-Verlag, Berlin, 1990.
7. R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
8. R. Alur, T. Feder, and T. A. Henzinger. The benefits of relaxing punctuality. *Journal of the ACM*, 43:116–146, 1996.
9. R. Alur, A. Itai, R. P. Kurshan, and M. Yannakakis. Timing verification by successive approximation. *Information and Computation*, 118(1):142–157, 1995.
10. R. Alur, R. Grosu, Y. Hur, V. Kumar, and I. Lee. Modular specification of hybrid systems in CHARON. In *HSCC*, pages 6–19, 2000.
11. E. Asarin and A. Bouajjani. Perturbed turing machines and hybrid systems. In *Proceedings of the 6th IEEE Symposium on Logic in Computer Science*, pages 269–278, 2001.
12. E. Asarin, T. Dang, and O. Maler. The **d/dt** tool for verification of hybrid systems. *Lecture Notes in Computer Science*, 2404:365–377, 2002.
13. E. Asarin, O. Maler, and A. Pnueli. On the analysis of dynamical systems having piecewise-constant derivatives. *Theoretical Computer Science*, 238:35–65, 1995.

14. E. Asarin, O. Maler, A. Pnueli, and J. Sifakis. Controller synthesis for timed automata. In *Proceedings of the IFAC Symposium on System Structure and Control*, pages 469–474. Elsevier, Amsterdam, 1998.
15. J. Bengtsson, W. O. D. Griffioen, K. J. Kristoffersen, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. Verification of an audio protocol with bus collision using UPPAAL. In R. Alur and T. A. Henzinger, editors, *CAV 96: Computer-aided Verification*, Lecture Notes in Computer Science 1102, pages 244–256. Springer-Verlag, Berlin, 1996.
16. J. Bengtsson, K. G. Larsen, F. Larsson, P. Pettersson, and W. Yi. UPPAAL: a tool-suite for automatic verification of real-time systems. In R. Alur, T. A. Henzinger, and E. D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 232–243. Springer-Verlag, Berlin, 1996.
17. F. Cassez, T. A. Henzinger, and J.-F. Raskin. A comparison of control problems for timed and hybrid systems. In *HSCC 02: Hybrid Systems—Computation and Control*, Lecture Notes in Computer Science 2289, pages 134–148. Springer-Verlag, Berlin, 2002.
18. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, Cambridge, Massachusetts, 1999.
19. E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal-logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, 1986.
20. P. Cousot. Abstract interpretation. *ACM Computing Surveys*, 28(2):324–328, June 1996.
21. C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In R. Alur, T.A. Henzinger, and E.D. Sontag, editors, *Hybrid Systems III*, Lecture Notes in Computer Science 1066, pages 208–219. Springer-Verlag, Berlin, 1996.
22. M. Fränzle. Analysis of hybrid systems: An ounce of realism can save an infinity of states. In *Proceedings of CSL'99: Computer Science Logic, LNCS 1683*, pages 126–140. Springer-Verlag, Berlin, 1999.
23. V. Gupta, T. A. Henzinger, and R. Jagadeesan. Robust timed automata. In *HART 97: Hybrid and Real-Time Systems*, Lecture Notes in Computer Science 1201, pages 331–345. Springer-Verlag, Berlin, 1997.
24. N. Halbwachs, P. Raymond, and Y.-E. Proy. Verification of linear hybrid systems by means of convex approximation. In B. LeCharlier, editor, *SAS 94: Static Analysis Symposium*, Lecture Notes in Computer Science 864, pages 223–237. Springer-Verlag, Berlin, 1994.
25. T. A. Henzinger. Sooner is safer than later. *Information Processing Letters*, 43:135–141, 1992.
26. T. A. Henzinger. Hybrid automata with finite bisimulations. In *ICALP 95: Automata, Languages, and Programming*, Lecture Notes in Computer Science 944, pages 324–335. Springer-Verlag, Berlin, 1995.
27. T. A. Henzinger. It's about time: Real-time logics reviewed. In *CONCUR 98: Concurrency Theory*, Lecture Notes in Computer Science 1466, pages 439–454. Springer-Verlag, Berlin, 1998.
28. T. A. Henzinger. The theory of hybrid automata. In M.K. Inan and R.P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, NATO ASI Series F: Computer and Systems Sciences 170, pages 265–292. Springer-Verlag, Berlin, 2000.
29. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. HYTECH: A model checker for hybrid systems. *Software Tools for Technology Transfer*, pages 110–122, 1997.

30. T. A. Henzinger, P.-H. Ho, and H. Wong-Toi. Algorithmic analysis of nonlinear hybrid systems. *IEEE Transactions on Automatic Control*, 43:540–554, 1998.
31. T. A. Henzinger, B. Horowitz, and R. Majumdar. Rectangular hybrid games. In *CONCUR 99: Concurrency Theory*, Lecture Notes in Computer Science 1664, pages 320–335. Springer-Verlag, Berlin, 1999.
32. T. A. Henzinger and P.W. Kopke. State equivalences for rectangular hybrid automata. In *CONCUR 96: Concurrency Theory*, Lecture Notes in Computer Science 1119, pages 530–545. Springer-Verlag, Berlin, 1996.
33. T. A. Henzinger and P. W. Kopke. Discrete-time control for rectangular hybrid automata. *Theoretical Computer Science*, 221:369–392, 1999.
34. T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata? *Journal of Computer and System Sciences*, 57:94–124, 1998.
35. T. A. Henzinger and R. Majumdar. A classification of symbolic transition systems. In *STACS 00: Theoretical Aspects of Computer Science*, Lecture Notes in Computer Science 1770, pages 13–34. Springer-Verlag, Berlin, 2000.
36. T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111:193–244, 1994.
37. T. A. Henzinger and J.-F. Raskin. Robust undecidability of timed and hybrid systems. In *HSCC 00: Hybrid Systems—Computation and Control*, Lecture Notes in Computer Science 1790, pages 145–159. Springer-Verlag, Berlin, 2000.
38. T. A. Henzinger, J.-F. Raskin, and P.-Y. Schobbens. The regular real-time languages. In *ICALP 98: Automata, Languages, and Programming*, Lecture Notes in Computer Science 1443, pages 580–591. Springer-Verlag, Berlin, 1998.
39. T. A. Henzinger and H. Wong-Toi. Using HyTech to synthesize control parameters for a steam boiler. In *Formal Methods for Industrial Applications: Specifying and Programming the Steam Boiler Control*, Lecture Notes in Computer Science 1165, pages 265–282. Springer-Verlag, Berlin, 1996.
40. M. W. Hirsh and S. Smale. *Differential Equations, Dynamical Systems and Linear Algebra*. Academic Press, 1974.
41. Y. Kesten, A. Pnueli, J. Sifakis, and S. Yovine. Integration graphs: a class of decidable hybrid systems. In R.L. Grossman, A. Nerode, A.P. Ravn, and H. Rischel, editors, *Hybrid Systems*, Lecture Notes in Computer Science 736, pages 179–208. Springer-Verlag, Berlin, 1993.
42. O. Maler, Z. Manna, and A. Pnueli. From timed to hybrid systems. In J.W. de Bakker, K. Huizing, W.-P. de Roever, and G. Rozenberg, editors, *Real Time: Theory in Practice*, Lecture Notes in Computer Science 600, pages 447–484. Springer-Verlag, Berlin, 1992.
43. R. Milner. An algebraic definition of simulation between programs. In *Second International Joint Conference on Artificial Intelligence*, pages 481–489. The British Computer Society, 1971.
44. A. Pnueli. The temporal logic of programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, pages 46–57. IEEE Computer Society Press, Washington D.C., 1977.
45. A. Puri and P. Varaiya. Decidability of hybrid systems with rectangular differential inclusions. In D.L. Dill, editor, *CAV 94: Computer-aided Verification*, Lecture Notes in Computer Science 818, pages 95–104. Springer-Verlag, Berlin, 1994.
46. J.-F. Raskin. *Logics, Automata, and Classical Theories for Deciding Real Time*. Ph.D. thesis, Facultés Universitaires Notre-Dame de la Paix, Namur, Belgium, 1999.

47. J.-F. Raskin and P.-Y. Schobbens. State-clock logic: a decidable real-time logic. In O. Maler, editor, *HART 97: Hybrid and Real-time Systems*, LNCS 1201, pages 33–47. Springer-Verlag, Berlin, 1997.
48. T. Stauner, O. Mueller, and M. Fuchs. Using HYTECH to verify an automotive control system. In O. Maler, editor, *Hybrid and Real-Time Systems*, LNCS 1201, pages 139–153, Grenoble, France. Springer Verlag, Berlin, 1997.
49. C. Tomlin. *Hybrid Control of Air Traffic Management Systems.* Ph.D. thesis, University of California at Berkeley, 1998.
50. H. Wong-Toi. The synthesis of controllers for linear hybrid automata. In *Proceedings of the 36th Conference on Decision and Control*, pages 4607–4612. IEEE Press, New-York, 1997.

An Overview of Hybrid Systems Control

John Lygeros

Department of Electrical and Computer Engineering,
University of Patras, Rio, Patras, GR 26500, Greece
lygeros@ee.upatras.gr

1 Introduction

The term hybrid systems is used in the literature to refer to systems that feature an interaction between diverse types of dynamics. Most heavily studied in recent years are hybrid systems that involve the interaction between continuous dynamics (with a dense state space and evolution determined by differential or difference equations) and discrete dynamics (with a finite or countable state space and evolution according to finite state machines, Petri nets, or other discrete models of computation). The study of this class of systems has to a large extent been motivated by applications to embedded systems and control. Embedded systems by definition involve the interaction of digital devices with a predominantly analog environment. In addition, much of the design complexity of embedded systems comes from the fact that they have to meet specifications such as hard real-time constraints, scheduling constraints, etc. that involve a mixture of discrete and continuous requirements. Therefore, both the model and the specifications of embedded systems can naturally be expressed in the context of hybrid systems. Motivated by the observation that embedded systems often also have to deal with an uncertain and potentially adversarial environment, researchers have in recent years extended their study of hybrid systems beyond continuous and discrete dynamics, to include probabilistic terms. This has led to the more general class of stochastic hybrid systems.

Control problems have been at the forefront of hybrid systems research from the very beginning. The reason is that many important applications with prominent hybrid dynamics come from the area of embedded control. For example, hybrid control has played an important role in applications to avionics [43,60], automated highways [34,41], automotive control [6], air traffic management [60, 61], industrial process control [23], and manufacturing and robotics [51, 57]. The corresponding problems in stochastic hybrid systems have found applications to insurance pricing [19], capacity expansion

models for the power industry [19], flexible manufacturing and fault tolerant control [26], and the control of communication networks [31].

The control problems that have arisen in these applications differ, first of all, in the way in which they treat uncertainty. Generally, the problems can be grouped into three classes:

1. Deterministic. Here it is assumed that there is no uncertainty; control inputs are the only class of inputs considered.
2. Non-deterministic. In this case inputs are grouped into two classes, control and disturbance. The design of a controller for regulating the control inputs assumes that disturbance inputs are adversarial. Likewise, the requirements are stated as worst case: the controller should be such that the specifications are met for all possible actions of the disturbance. From a control perspective, problems in this class are typically framed in the context of robust control, or game theory.
3. Stochastic. Again, both control and disturbance inputs are considered. The difference with the non-deterministic case is that a probability distribution is assumed for the disturbance inputs. This extra information can be exploited by the controller and also allows one to formulate finer requirements. For example, it may not be necessary to meet the specifications for all disturbances, as long as the probability of meeting them is high enough.

In addition, the control problems studied in the literature differ in the specifications they try to meet. Generally, according to the specification the problems can also be grouped into three classes:

1. Stabilization. Here the problem is to select the continuous inputs and/or the timing and destinations of discrete switches to make sure that the system remains close to an equilibrium point, limit cycle, or other invariant set. Many variants of this problem have been studied in the literature. They differ in the type of control inputs considered (discrete, continuous, or both) and the type of stability specification (stabilization, asymptotic or exponential stabilization, practical stabilization, etc.). Even more variants have been considered in the case of stochastic hybrid systems (stability in distribution, moment stability, almost sure asymptotic stability, etc.).
2. Optimal control. Here the problem is to steer the hybrid system using continuous and/or discrete controls in a way that minimizes a certain cost function. Again, different variants have been considered, depending on whether discrete and/or continuous inputs are available, whether cost is accumulated along continuous evolution and/or during discrete transitions, whether the time horizon over which the optimization is carried out is finite or infinite, etc.
3. Language specifications. Control problems of great interest can also be formulated by imposing the requirement that the trajectories of the closed-loop system are all contained in a set of desirable trajectories. Typical

requirements of this type arise from reachability considerations, either of the safety type (along all trajectories the state of the system should remain in a "good" region of the state space), or of the liveness type (the state of the system should eventually reach a "good" region of the state space along all trajectories). Starting with these simple requirements, progressively more and more complex specifications can be formulated: the state should visit a given set of states infinitely often, given two sets of states, if the state visits one infinitely often it should also visit the other infinitely often, etc. These specifications are all related to the "language" generated by the closed-loop system and have been to a large extent motivated by analogous problems formulated for discrete systems based on temporal logic [46].

In this chapter we provide an introduction to the problems addressed in all these areas. In Section 3 we formulate a number of hybrid stabilization problems, state the main approaches to solving these problems, and provide references to publications where more details can be found. In Sections 4 and 5 we do the same with optimal control problems and language specification problems, respectively. In each of these three sections emphasis is placed separately on the three types of disturbances: deterministic, non-deterministic, and stochastic.

To be able to clearly state the different control problems of interest, we start by introducing a simple hybrid system model (Section 2). We stress that this hybrid model is meant to be used only for illustration purposes. It is not the model used in any of the references, nor does it claim to be a general model for controlled hybrid systems. Its one advantage is that it is simple enough to be understood by the non-specialist but also general enough to be used to formulate most of the control problems of interest (with the notable exception of stochastic control problems, whose formulation requires considerably more mathematical sophistication). More general and appropriate models can be found in the references and other chapters of this volume.

2 A Simple Hybrid Control Model

Hybrid control problems have been formulated for both continuous- and discrete-time systems. As usual, continuous-time problems present more technical difficulties. In this section we introduce a model suitable for formulating continuous-time control problems for deterministic hybrid systems. We also discuss briefly the simplifications that arise if discrete-time systems are considered and the complications involved in extending the model to stochastic systems. As discussed above the model introduced here is "minimalist" in the sense that it includes the minimum set of features necessary to clarify the distinctions among the different control problems considered in subsequent sections.

2.1 Syntax: Non-deterministic systems

Since we are interested in hybrid dynamics, the dynamical systems we consider involve both a continuous state (denoted by x) and a discrete state (denoted by q). To allow us to capture the different types of uncertainties discussed above, we also assume that the evolution of the state is influenced by two different kinds of inputs: controls and disturbances. We assume that inputs of each kind can be either discrete or continuous, and we use υ to denote discrete controls, u to denote continuous controls, δ to denote discrete disturbances, and d to denote continuous disturbances.

The dynamics of the state are determined through four functions: a vector field f that determines the continuous evolution, a reset map r that determines the outcome of the discrete transitions, a "guard" set that determines when discrete transitions can take place, and a "domain" set Dom that determines when continuous evolution is possible. The following definition formalizes the details.

Definition 1 (Hybrid game automaton). *A hybrid game automaton (HGA) characterizes the evolution of*

- *discrete state variables $q \in Q$ and continuous state variables $x \in X$,*
- *discrete control inputs $\upsilon \in \Upsilon$ and continuous control inputs $u \in U$ and*
- *discrete disturbance inputs $\delta \in \Delta$ and continuous disturbance inputs $d \in D$*

by means of four functions

- *a vector field $f : Q \times X \times U \times D \to X$,*
- *a domain set $Dom : Q \times \Upsilon \times \Delta \to 2^X$,*
- *guard sets $G : Q \times Q \times \Upsilon \times \Delta \to 2^X$, and*
- *a reset function $r : Q \times Q \times X \times U \times D \to X$.*

As usual, 2^X stands for the set of all subsets (power set) of X; in other words, Dom and G are set-valued maps. For simplicity, we assume that $X = \mathbb{R}^n$, $U \subseteq \mathbb{R}^m$, and $D \subseteq \mathbb{R}^p$ for integers n, m, and p. A similar definition can also be formulated for discrete-time hybrid systems, simply by considering f as a transition function rather than as a vector field. In this case the discrete-time hybrid system can be considered as a simple discrete-time system, with state space $Q \times X$ and a set-valued transition relation

$$R(q, x, u, d, \upsilon, \delta) =$$
$$= [\{q\} \times f(q, x, u, d)] \cup \left[\bigcup_{\{q' \in Q \,:\, x \in G(q,q',\upsilon,\delta)\}} \{q'\} \times r(q, q', x, u, d) \right],$$

if $x \in \mathrm{Dom}(q, \upsilon, \delta)$ and

$$\bigcup_{\{q' \in Q \,:\, x \in G(q,q',\upsilon,\delta)\}} \{q'\} \times r(q, q', x, u, d)$$

otherwise. Even though this abstraction appears convenient and is suitable for certain classes of problems, it is often desirable to exploit additional structure by developing more detailed (rather than more abstract) models of discrete-time hybrid systems.

To avoid pathological situations (lack of solutions, deadlock, chattering, etc.) one needs to introduce technical assumptions on the model components. Typically, these include continuity assumptions on f and r, compactness assumptions on U and D, and convexity assumptions on $\bigcup_{u\in U} f(q,x,u,d)$ and $\bigcup_{d\in D} f(q,x,u,d)$, etc. These assumptions aim to ensure, among other things, that for all $q \in Q$, $x_0 \in X$ and $u(\cdot)$, $d(\cdot)$ measurable functions of time, the differential equation

$$\dot{x}(t) = f(q, x(t), u(t), d(t))$$

has a unique solution $x(\cdot) : \mathbb{R}_+ \to X$ with $x(0) = x_0$. Additional assumptions are often imposed to prevent deadlock, a situation where it is not possible to proceed by continuous evolution or by discrete transition. A typical assumption to prevent this situation is that the set $\mathrm{Dom}(q, \upsilon, \delta)$ is open and if $x \notin \mathrm{Dom}(q, \upsilon, \delta)$ then $x \in \bigcup_{q'\in Q} G(q, q', \upsilon, \delta)$. Finally, in many publications assumptions are introduced to prevent what is called the Zeno phenomenon, a situation where the solution of the system takes an infinite number of discrete transitions in a finite amount of time. The Zeno phenomenon can prove particularly problematic for hybrid control problems, since it may be exploited either by the control or by the disturbance variables. For example, a controller may appear to meet a safety specification by forcing all trajectories of the system to be Zeno; [37,61] provide examples of this situation that arise in the deterministic water tank benchmark problem and in a non-deterministic collision avoidance problem from air traffic management. This situation is undesirable in practice, since the specifications are met not because of successful controller design but because of modeling over-abstraction. In addition, Zeno controllers require infinitely fast switching and cannot be implemented in practice. For these reasons, the Zeno phenomenon is usually forbidden by direct assumptions. In some cases, structural assumptions are introduced on the model to prevent Zeno solutions (e.g., by enforcing a lower bound on the time between discrete transitions or the time to traverse each discrete state cycle).

Many of the assumptions discussed here can be relaxed, replaced by other variants, or dropped altogether; for example, if we consider relaxed controls in optimal control problems, convexity and compactness assumptions are typically not needed. For discrete-time hybrid systems, most of these assumptions are unnecessary. For example, deadlock and the Zeno phenomenon are typically not issues for discrete-time systems.

2.2 Example: A four-gear car

Fig. 1 shows a simplified discrete model of a car with a gear box having four gears; transitions between any gears are allowed in practice, but they are

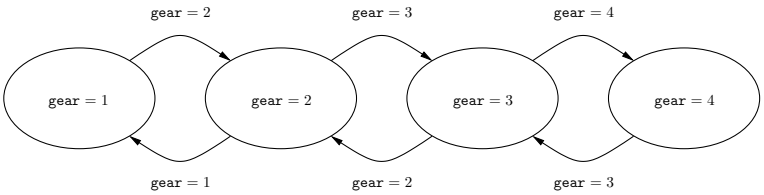


Fig. 1. A hybrid system modeling a car with four gears

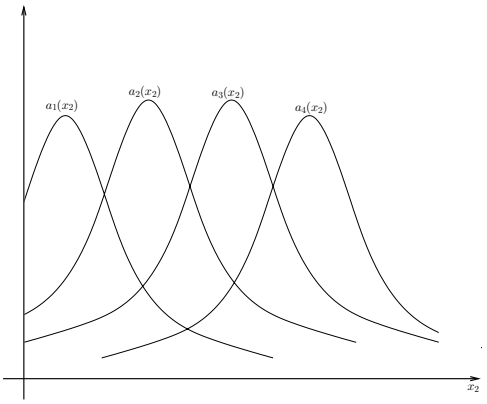


Fig. 2. The efficiency functions of the different gears

omitted in the figure to keep it simple. We would like to couple this model with a continuous model to capture the longitudinal motion of the car along the road; lateral dynamics will be ignored.

Let x_1 denote the longitudinal position of the car and let x_2 denote its velocity. The model has two control inputs: the gear selection, denoted by $\texttt{gear} \in \{1, \ldots, 4\}$, and the throttle position, denoted by $u \in [u_{\min}, u_{\max}]$. Gear shifting is necessary because little power can be generated by the engine at very low or very high engine speeds. The function α_i represents the efficiency of gear i as a function of speed. Typical shapes of the functions α_i are shown in Fig. 2. In addition to the controls, the model also has a disturbance input, $d \in [-\theta_{\min}, \theta_{\max}]$ which represents the inclination of the road.

In our context, the four-gear car can be modeled by an HGA with four discrete states ($q \in \{1, 2, 3, 4\} = Q$, one for each gear), two continuous states (the position and velocity of the vehicle, $x \in \mathbb{R}^2 = X$), one discrete input ($\upsilon \in \{1, 2, 3, 4\} = \Upsilon$, representing the gear shift commands), one continuous input ($u \in [u_{\min}, u_{\max}]$, representing the engine throttle command), and one continuous disturbance ($d \in [-\theta_{\max}, \theta_{\max}]$, representing road inclination). The continuous dynamics are governed by the vector field

$$f(q,x,u,d) = \begin{bmatrix} x_2 \\ a_q(x_2)u + mg\sin(d) \end{bmatrix},$$

and the domain of continuous evolution is given by

$$\mathrm{Dom}(q,\upsilon) = \begin{cases} \mathbb{R}^2 & \text{if } \upsilon = q \\ \emptyset & \text{otherwise.} \end{cases}$$

This forces discrete transitions to take place whenever the driver requests a gear switch and allows the discrete state to remain the same otherwise. The guard sets are given by

$$G(q,q',\upsilon) = \begin{cases} \mathbb{R}^2 & \text{if } \upsilon = q' \text{ and } q \neq q' \\ \emptyset & \text{otherwise.} \end{cases}$$

Finally, the reset function leaves the continuous state unchanged, i.e.,

$$r(q,q',x,u,d) = x.$$

Several interesting control problems can be formulated for this simple car model. For example, what is the optimal control strategy to drive between two points in minimum time? The problem is not trivial if we include the reasonable assumption that each gear shift takes a certain amount of time. The optimal controller, which can be modeled as a hybrid system, may be derived using the theory of optimal control of hybrid systems.

2.3 Semantics: Solutions or runs

To formally define the solutions of this class of hybrid systems, we recall the following notion from [42].

Definition 2 (Hybrid time set). *A hybrid time set $\tau = \{I_i\}_{i=0}^{N}$ is a finite or infinite sequence of intervals of the real line, such that*

- *for all $i < N$, $I_i = [\tau_i, \tau_i']$;*
- *if $N < \infty$, then either $I_N = [\tau_N, \tau_N']$, or $I_N = [\tau_N, \tau_N')$, possibly with $\tau_N' = \infty$;*
- *for all i, $\tau_i \leq \tau_i' = \tau_{i+1}$.*

Since the dynamical systems considered here are time invariant, without loss of generality we can assume that $\tau_0 = 0$. It easy to see that, although more complicated than the usual time sets (the real numbers for continuous-time systems or the integers for discrete-time systems), hybrid time sets are reasonably well-behaved mathematical objects. For example, each hybrid time set is totally ordered, whereas the set of all hybrid time sets is partially ordered. One can therefore naturally define prefixes and suffixes of a hybrid time set, maximal elements of a collection of hybrid time sets, etc. For discrete-time

hybrid systems, the introduction of hybrid time sets is unnecessary, since the set of integers or natural numbers can typically be used.

Roughly speaking, the solution of an HGA (often called a "run" or an "execution") is defined over a hybrid time set τ and involves a sequence of intervals of continuous evolution followed by discrete transitions. Starting at some initial state (q_0, x_0) the continuous state moves along the solution of the differential equation

$$\dot{x} = f(q_0, x, u, d)$$

as long as it does not leave the set $\mathrm{Dom}(q_0, \upsilon, \delta)$. The discrete state remains constant throughout this time. If at some point x reaches a set $G(q_0, q', \upsilon, \delta)$ for some $q' \in Q$, a discrete transition can take place. The first interval of τ ends and the second one begins with a new state (q', x') where x' is determined by the reset map r. The process is then repeated. Notice that considerable freedom is allowed when defining the solution in this "declarative" way: in addition to the effect of the input variables, there may also be a choice between evolving continuously or taking a discrete transition (if the continuous state is in both the domain set and a guard set) or between multiple discrete transitions (if the continuous state is in many guard sets at the same time).

The following concept helps to formalize the above discussion.

Definition 3 (Hybrid trajectory). *Given a set of variables, a, that take values in a set A, a hybrid trajectory over this set of variables is a pair (τ, a) where $\tau = \{I_i\}_{i=0}^N$ is a hybrid time set and $a = \{a_i(\cdot)\}_{i=0}^N$ is a sequence of functions $a_i(\cdot) : I_i \to A$.*

The solutions of the HGA can now be defined as hybrid trajectories over its state and input variables.

Definition 4 (Run). *A run of an HGA is a hybrid trajectory $(\tau, q, x, \upsilon, u, \delta, d)$ over its state and input variables that satisfies the following conditions:*

- *Discrete evolution: for $i < N$,*
  1. $x_i(\tau_i') \in G(q_i(\tau_i'), q_{i+1}(\tau_{i+1}), \upsilon_i(\tau_i'), \delta_i(\tau_i'))$.
  2. $x_{i+1}(\tau_{i+1}) = r(q_i(\tau_i'), q_{i+1}(\tau_{i+1}), x_i(\tau_i'), u_i(\tau_i'), d_i(\tau_i'))$.
- *Continuous evolution: for all i with $\tau_i < \tau_i'$*
  1. *$u_i(\cdot)$ and $d_i(\cdot)$ are measurable functions.*
  2. *$q_i(t) = q_i(\tau_i)$ for all $t \in I_i$.*
  3. *$x_i(\cdot)$ is a solution of the differential equation*

$$\dot{x}_i(t) = f(q_i(t), x_i(t), u_i(t), d_i(t))$$

  over the interval I_i starting at $x_i(\tau_i)$.
  4. *$x_i(t) \in Dom(q_i(t), \upsilon_i(t), \delta_i(t))$ for all $t \in [\tau_i, \tau_i')$.*

For discrete-time hybrid systems the definition of a run is again much simpler. A run can simply be defined as a finite or infinite sequence of states and inputs, $\{q_i, x_i, u_i, d_i, \upsilon_i, \delta_i\}_{i=0}^N$, such that for all i

$$(q_{i+1}, x_{i+1}) \in R(q_i, x_i, u_i, d_i, \upsilon_i, \delta_i).$$

2.4 Classification of control action

The preceding model allows control and disturbance inputs to influence the evolution of the system in a number of ways. In particular, control and disturbance can

1. Steer the continuous evolution through the effect of u and d on the vector field f.
2. Force discrete transitions to take place through the effect of υ and δ on the domain Dom.
3. Affect the discrete state reached after a discrete transition through the effect of υ and δ on the guards G.
4. Affect the continuous state reached after a discrete transition through the effect of u and d on the reset function r.

Notice that the model implicitly restricts the influence of the discrete inputs υ and δ to the timing and discrete destination of discrete transitions and the influence of the continuous inputs u and d to continuous evolution and the continuous destination of discrete transitions. At this level of generality all inputs could, in fact, be allowed to influence all aspects of the evolution of the system. Caution should be taken, however, when doing this, since experience suggests that it tends to severely complicate the technicalities associated with the definition of runs, ensuring that runs exist for all inputs, preventing chattering strategies, etc. Experience also suggests that this type of mixing of discrete and continuous inputs is rarely needed in practice.

Another issue that arises is the type of controllers one allows for selecting the control inputs u and υ. The most common control strategies considered in the hybrid systems literature are, of course, static feedback strategies. In this case the controller can be thought of as a map (in general set valued) of the form

$$g : Q \times X \to 2^{\Upsilon \times U}.$$

For controllers of this type, the runs of the closed-loop system can easily be defined as runs, $(\tau, q, x, \upsilon, u, \delta, d)$, of the uncontrolled system such that for all $I_i \in \tau$ and all $t \in I_i$

$$(\upsilon_i(t), u_i(t)) \in g(q_i(t), x_i(t)).$$

It turns out that for certain kinds of control problems (for example, reachability problems) one can restrict attention to feedback controllers without loss of generality. For other problems, however, one may be forced to consider more general classes of controllers: dynamic feedback controllers that incorporate observers for output feedback problems, controllers that involve nonanticipative strategies for gaming problems, piecewise constant controllers to prevent chattering, etc. Even for these types of controllers, it is usually intuitively clear what one means by the runs of the closed-loop system. However, unlike feedback controllers, a formal definition would require one to formulate the problem in a compositional hybrid systems framework and formally

define the closed-loop system as the composition of a plant and a controller automaton.

2.5 Stochastic hybrid systems

The controlled hybrid system model presented above allows one to capture a number of interesting and important hybrid phenomena. Many of the deterministic and non-deterministic hybrid control problems considered in the literature can be recast in this framework. The model, however, does not contain any stochastic terms. The formal definition of stochastic hybrid models requires considerable mathematical overhead, even in the simplest cases. Here we briefly describe the types of stochastic phenomena that can appear in hybrid systems, only to familiarize the reader with the issues that arise; more details can of course be found in the references.

Stochastic terms can enter hybrid dynamics in a number of different places:

1. Continuous evolution may be governed by stochastic differential equations.
2. Discrete transitions may take place spontaneously, at a given, possibly state-dependent, rate (as they do for example in discrete Markov chains). Some authors also consider forced transitions, which take place whenever the continuous state tries to leave a given set (the equivalent of the Dom set introduced above).
3. The destination of discrete transitions may be given by a probability kernel.

As for deterministic and non-deterministic systems, one can also consider controls that influence the same places: for example, controls that steer continuous evolution through controlled diffusions, influence the rate at which discrete transitions take place, determine the boundaries at which they are forced, or influence the probability distribution that determines the destination of discrete transitions.

Clearly, all these alternatives allow for the formulation of countless variants of control problems. The literature on the control of stochastic hybrid systems is therefore diverse. In Table 1 we summarize the modeling choices made in some of the references listed in subsequent sections.

3 Stabilization of Hybrid Systems

The problem of stabilizing hybrid systems is designing controllers such that the runs of the closed-loop system remain close and possibly converge to a given invariant set. An invariant set is a set of states with the property that runs starting in the set remain in the set forever. More formally, $W \subseteq Q \times X$ is an invariant set if for all $(\hat{q}, \hat{x}) \in W$ and all runs $(\tau, q, x, \upsilon, u, \delta, d)$ starting at $(\hat{q}, \hat{x})$,

$$(q_i(t), x_i(t)) \in W, \ \ \forall I_i \in \tau, \ \forall t \in I_i.$$

Table 1. Overview of stochastic hybrid models

Characteristics	[18, 19]	[26, 27]	[11, 49]	[47, 68]	[31]	[36]	[15]
Stochastic continuous evolution		√	√	√		√	√
Forced transitions	√		√			√	√
Spontaneous transitions	√	√		√	√		√
Probabilistic state reset	√		√		√	√	√
Continuous control	√	√	√	√			
Transition rate control	√	√					
Forcing control	√		√				
Continuous reset control	√		√				

The most common invariant sets are those associated with equilibria, points $\hat{x} \in X$ that are preserved under both discrete and continuous evolution, i.e.,

$$f(q, \hat{x}, u, d) = 0 \text{ and } r(q, q', \hat{x}, u, d) = \hat{x}$$

for all $q, q' \in Q$. An equilibrium $\hat{x}$ naturally defines an invariant set $Q \times \{\hat{x}\}$.

The definitions of stability can naturally be extended to hybrid systems by defining a metric on the hybrid state space. An easy way to do this is to consider the Euclidean metric on the continuous space and the discrete metric on the discrete space ($d_D(q, q') = 0$ if $q = q'$ and $d_D(q, q') = 1$ if $q \neq q'$) and define the hybrid metric by

$$d_H((q, x), (q', x')) = d_D(q, q') + \|x - x'\|.$$

The metric notation can be extended to sets in the usual way:

$$d_H((q, x), W) = \inf_{(q'x') \in W} d_H((q, x), (q', x')).$$

Equipped with this metric, the standard stability definitions (Lyapunov stability, asymptotic stability, exponential stability, practical stability, etc.) naturally extend from the continuous to the hybrid domain. For example, an invariant set, W, is called stable if for all $\epsilon > 0$ there exists $\epsilon' > 0$ such that for all $(q, x) \in Q \times X$ with $d_H((q, x), W) < \epsilon'$ and all runs $(\tau, q, x, \upsilon, u, \delta, d)$ starting at (q, x),

$$d_H((q_i(t), x_i(t)), W) < \epsilon, \ \ \forall I_i \in \tau, \ \forall t \in I_i.$$

Stability of hybrid systems has been extensively studied in recent years (see the overview papers [22, 40]). By comparison, the work on stabilization problems is relatively sparse. A family of stabilization schemes assumes that the continuous dynamics are given, for example, stabilizing controllers have been designed for each $f(q,\cdot,\cdot,\cdot)$. Procedures are then defined for determining the switching times (or at least constraints on the switching times) to ensure that the closed-loop system is stable, asymptotically stable, or practically stable [32, 55, 62, 66]. Stronger results are possible for special classes of systems, such as planar systems [35, 64]. For non-deterministic systems, in [25] an approach to the practical exponential stabilization of a class of hybrid systems with disturbances is presented. For a brief overview of stabilization problems for stochastic hybrid systems the reader is referred to [67].

4 Optimal Control of Hybrid Systems

In optimal control problems it is typically assumed that a cost is assigned to the different runs of the hybrid system by means of a cost function. The objective of the controller is then to minimize this cost among all possible runs by selecting the values of the control variables appropriately. Typically, the cost function assigns a cost to both continuous evolution and discrete transitions. For example, for the cost assigned to a run $(\tau, q, x, \upsilon, u, \delta, d)$ with $\tau = \{I_i\}_{i=0}^{N}$, the cost function may have the form

$$\sum_{i=0}^{N} \Big[\int_{\tau_i}^{\tau_i'} l(q_i(t), x_i(t), u_i(t), d_i(t))dt + g(q_i(\tau_i'), x_i(\tau_i'), q_{i+1}(\tau_{i+1}'), x_{i+1}(\tau_{i+1}), u_i(\tau_i), d_i(\tau_i), \upsilon_i(\tau_i'), \delta_i(\tau_i')) \Big],$$

where $l : Q \times X \times U \times D \to \mathbb{R}$ is a function assigning a cost to the pieces of continuous evolution and $g : Q \times X \times Q \times X \times U \times D \times \Upsilon \times \Delta \to \mathbb{R}$ is a function assigning a cost to discrete transitions. Different variants of optimal control problems can be formulated, depending on, e.g., the type of cost function, the horizon over which the optimization takes place (finite or infinite), or whether the initial and/or final states are specified.

As with continuous systems, two different approaches have been developed for addressing such optimal control problems. One is based on the maximum principle and the other on dynamic programming. Extensions of the maximum principle to hybrid systems have been proposed by numerous authors; see, for example, [28, 52, 56, 58]. Computational tools based on this theory have also been developed [56, 65]. The solution of the optimal control problem with the dynamic programming approach typically requires the computation of a value function, which is characterized as a viscosity solution to a set of variational or quasi-variational inequalities [10, 14]. This approach has also been extended to classes of stochastic hybrid systems; see, for example, [11, 24, 26]. Computational methods for solving the resulting variational and quasi-variational

inequalities are presented in [7, 16, 50]. For simple classes of systems (e.g., timed automata [1]) and simple cost functions (e.g., minimum time problems) it is often possible to exactly compute the optimal cost and optimal control strategy, without resorting to numerical approximations; see, for example, [2, 4, 12, 45].

A somewhat different optimal control problem arises when one tries to control hybrid systems using model predictive or receding horizon techniques. Generally, the aim here is to use a model to predict the future evolution of the system under different inputs and then employ optimization algorithms to select the inputs that promise the "best" future. The initial part of these inputs is applied to the system, a new measurement is taken (providing feedback), and the process is repeated. For hybrid systems, such a model predictive control approach has primarily been studied in discrete time; see, for example, [8, 48]. The toolbox of [39] provides functions for the numerical solution of hybrid model predictive control problems (and much more).

5 Language Specification Problems

Another type of control problem that has attracted considerable attention in the hybrid systems literature revolves around language specifications. One example of language specifications is the *safety specifications*. In this case a "good" set of states $W \subseteq Q \times X$ is given and the designer is asked to produce a controller that ensures that the state always stays in this set; in other words, for all runs $(\tau, q, x, \upsilon, u, \delta, d)$ of the closed-loop system

$$\forall I_i \in \tau \;\; \forall t \in I_i, \;\; (q_i(t), x_i(t)) \in W.$$

The name "safety specifications" (which is given a formal meaning in computer science) intuitively refers to the fact that such specifications can be used to encode safety requirements in a system to ensure that nothing bad happens, e.g., in an air traffic management system to ensure that aircraft do not come closer to one another than a certain minimum distance.

Safety specifications are usually easy to meet (e.g., if aircraft never take off, mid-air collisions are impossible). To make sure that in addition to being safe the system actually does something useful, liveness specifications are usually also imposed. The simplest type of *liveness specification* deals with reachability: given a set of states $W \subseteq Q \times X$, design a controller such that for all runs $(\tau, q, x, \upsilon, u, \delta, d)$ of the closed-loop system

$$\exists I_i \in \tau \;\; \exists t \in I_i, \;\; (q_i(t), x_i(t)) \in W.$$

In the air traffic context a minimal liveness type requirement is to make sure that aircraft eventually arrive at their destinations. Mixing different types of specifications like the ones given above one can construct arbitrarily complex properties, e.g., ensure that the state visits a set infinitely often, ensure that

it reaches a set and stays there forever after, etc. Such complex *language specifications* are usually encoded formally using temporal logic notation [46].

Controller design problems under language specifications have been studied very extensively for discrete systems in the computer science literature, mostly under the name *synthesis problems* (see [59] for an overview). More recently, a control perspective was given on this topic by Ramadge, Wonham, and co-workers [53]. The approach was then extended to classes of hybrid systems such as timed automata (systems with continuous dynamics of the form $\dot{x} = 1$, [5,33]) and rectangular automata (systems with continuous dynamics of the form $\dot{x} \in [l, u]$ for fixed parameters l, u, [63]). For systems of this type, exact and automatic computation of the controllers may be possible using model checking tools [9, 21, 30]. In all these cases the controller affects only the discrete aspects of the system evolution, i.e., the destination and timing of discrete transitions. More general language problems (e.g., where the dynamics are linear, the controller affects the continuous motion of the system) can be solved automatically in discrete time using methods from mathematical programming [39].

Extensions to general classes of hybrid systems in continuous time have been concerned primarily with computable numerical approximations of reachable sets using polyhedral approximations [3,17,29,54], ellipsoidal approximations [13], or more general classes of sets (e.g., defined using the solutions of the continuous system [20,38]). A useful link in this direction has been the relation between reachability problems and optimal control problems with an l_∞ penalty function [44,60]. This link has allowed the development of numerical tools that use partial differential equation solvers to approximate the value function of the optimal control problems and hence indirectly characterize reachable sets [50].

6 Concluding Remarks and Open Problems

The topic of hybrid control has attracted considerable attention from the research community in recent years. This has produced a number of theoretical and computational methods, which are now available to the designer and have been used successfully in a wide range of applications. There are still, however, many details that need to be clarified, as well as substantial problems that have not been studied in sufficient detail. We conclude this overview by listing some of these problems (by no means an exhaustive list).

A number of interesting problems arise in the area of dynamic feedback, which is still unexplored to a large extent. The rapid development in the design of hybrid observers witnessed in recent years poses the question of how the system will perform if the state estimates that the observers produce are used in state feedback. General principles (like the separation principle in linear systems) are probably too much to hope for in a general hybrid setting, but substantial progress may still be possible for specific subclasses.

A second area that, despite numerous contributions, still poses formidable problems is the area of hybrid games. As in the robust control of continuous systems, gaming appears in hybrid systems when one adopts a nondeterministic point of view to the control of uncertain systems. Unlike continuous systems, however, even fundamental notions such as "information" and "strategy" are still the topic of debate in hybrid systems. It is hoped that advances in this front will eventually lead to a robust control theory for classes of uncertain hybrid systems.

Finally, stochastic hybrid systems pose a number of challenges. For example, the formulation and solution of language specifications (even of the simplest safety type) for stochastic hybrid systems is still to a large extent open. Progress in this area could come by blending results for stochastic discrete event systems with results on the l_∞ optimal control of stochastic systems.

Acknowledgments

The writing of this overview was supported by the European Commission under the project COLUMBUS, IST-2001-38314, and the Network of Excellence HyCon, IST-511368.

References

1. R. Alur and D. Dill. A theory of timed automata. *Theoretical Computer Science*, 126:183–235, 1994.
2. R. Alur, S. La Torre, and G.J. Pappas. Optimal paths in weighted timed automata. In M. Di Benedetto and A. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control*, number 2034 in LNCS, pages 49–62. Springer-Verlag, Berlin, 2001.
3. E. Asarin, O. Bournez, T. Dang, O. Maler, and A. Pnueli. Effective synthesis of switching controllers for linear systems. *Proceedings of the IEEE*, 88(7):1011–1025, July 2000.
4. E. Asarin and O. Maler. As soon as possible: Time optimal control for timed automata. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, number 1569 in LNCS, pages 19–30. Springer-Verlag, Berlin, 1999.
5. E. Asarin, O. Maler, and A. Pnueli. Symbolic controller synthesis for discrete and timed systems. In P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry, editors, *Proceedings of Hybrid Systems II*, number 999 in LNCS, pages 1–20. Springer-Verlag, Berlin, 1995.
6. A. Balluchi, L. Benvenuti, M.D. Di Benedetto, C. Pinello, and A.L. Sangiovanni-Vincentelli. Automotive engine control and hybrid systems: Challenges and opportunities. *Proceedings of the IEEE*, 88(7):888–912, July 2000.
7. M. Bardi and I. Capuzzo-Dolcetta. *Optimal Control and Viscosity Solutions of Hamilton–Jacobi–Bellman Equations*. Birkhäuser, Boston, MA, 1997.

8. A. Bemporad and M. Morari. Control of systems integrating logic dynamics and constraints. *Automatica*, 35(3):407–427, March 1999.
9. J. Bengtsson, K.G. Larsen, F. Larsson, P. Petterson, and W. Yi. UPAAL: A tool suit for automatic verification of real-time systems. In *Hybrid Systems III*, number 1066 in LNCS, pages 232–243. Springer-Verlag, Berlin, 1996.
10. A. Bensoussan and J.L. Menaldi. Hybrid control and dynamic programming. *Dynamics of Continuous, Discrete and Impulsive Systems*, (3):395–442, 1997.
11. A. Bensoussan and J.L. Menaldi. Stochastic hybrid control. *Journal of Mathematical Analysis and Applications*, 249:261–288, 2000.
12. G. Berhmann, A. Fehnker, T. Hune, K.G. Larsen, P. Pettersson, J. Romijn, and F. Vaandrager. Minimum cost reachability for priced timed automata. In M. Di Benedetto and A. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control*, number 2034 in LNCS, pages 147–161. Springer-Verlag, Berlin, 2001.
13. O. Botchkarev and S. Tripakis. Verification of hybrid systems with linear differential inclusions using ellipsoidal approximations. In Nancy Lynch and Bruce H. Krogh, editors, *Hybrid Systems: Computation and Control*, number 1790 in LNCS, pages 73–88. Springer-Verlag, Berlin, 2000.
14. M.S. Branicky, V.S. Borkar, and S.K. Mitter. A unified framework for hybrid control: Model and optimal control theory. *IEEE Transactions on Automatic Control*, 43(1):31–45, 1998.
15. M. Bujorianu. Extended stochastic hybrid systems and their reachability problem. In R. Alus and G.J. Pappas, editors, *Hybrid Systems: Computation and Control*, number 2993 in LNCS, pages 234–249. Springer-Verlag, Berlin, 2004.
16. P. Cardaliaguet, M. Quincampoix, and P. Saint-Pierre. Numerical schemes for discontinuous value functions of optimal control. *Set Valued Analysis*, 8:111–126, 2000.
17. A. Chutinam and B. Krogh. Verification of polyhedral-invariant hybrid automata using polygonal flow pipe approximations. In Frits W. Vaandrager and Jan H. van Schuppen, editors, *Hybrid Systems: Computation and Control*, number 1569 in LNCS, pages 76–90. Springer-Verlag, Berlin, 1999.
18. M.H.A. Davis. Piecewise-deterministic Markov processes: A general class of non-diffusion stochastic models. *Journal of the Royal Statistical Society, B*, 46(3):353–388, 1984.
19. M.H.A. Davis. *Markov Processes and Optimization*. Chapman & Hall, London, 1993.
20. J.M. Davoren and A. Nerode. Logics for hybrid systems. *Proceedings of the IEEE*, 88(7):985–1010, July 2000.
21. C. Daws, A. Olivero, S. Trypakis, and S. Yovine. The tool KRONOS. In R. Alur, T. Henzinger, and E. Sontag, editors, *Hybrid Systems III*, number 1066 in LNCS, pages 208–219. Springer-Verlag, Berlin, 1996.
22. R. De Carlo, M. Branicky, S. Pettersson, and B. Lennarston. Perspectives and results on the stability and stabilizability of hybrid systems. *Proceedings of the IEEE*, 88(7):1069–1082, July 2000.
23. S. Engell, S. Kowalewski, C. Schulz, and O. Stursberg. Continuous-discrete interactions in chemical processing plants. *Proceedings of the IEEE*, 88(7):1050–1068, July 2000.
24. M. Farid. *Optimal Control, Piecewise Deterministic Processes and Viscosity Solutions*. Ph.D. thesis, Department of Electrical and Electronic Engineering, Imperial College of Science, Technology and Medicine, London, 1997.

25. Y. Gao, J. Lygeros, M. Quincampoix, and N. Seube. On the control of uncertain impulsive systems: Approximate stabilization and controlled invariance. *International Journal of Control*, 2005. To appear.
26. M.K. Ghosh, A. Arapostathis, and S.I. Marcus. Optimal control of switching diffusions with application to flexible manufacturing systems. *SIAM Journal on Control Optimization*, 31(5):1183–1204, September 1993.
27. M.K. Ghosh, A. Arapostathis, and S.I. Marcus. Ergodic control of switching diffusions. *SIAM Journal on Control Optimization*, 35(6):1952–1988, November 1997.
28. G. Grammel. Maximum principle for a hybrid system via singular pertrubations. *SIAM Journal of Control and Optimization*, 37(4):1162–1175, 1999.
29. M.R. Greenstreet and I. Mitchell. Integrating projections. In S. Sastry and T.A. Henzinger, editors, *Hybrid Systems: Computation and Control*, number 1386 in LNCS, pages 159–174. Springer-Verlag, Berlin, 1998.
30. T. A. Henzinger, P. H. Ho, and H. Wong Toi. A user guide to HYTECH. In E. Brinksma, W. Cleaveland, K. Larsen, T. Margaria, and B. Steffen, editors, *TACAS 95: Tools and Algorithms for the Construction and Analysis of Systems*, number 1019 in LNCS, pages 41–71, Springer-Verlag, Berlin, 1995.
31. J. Hespanha. Stochastic hybrid systems: Application to communication networks. In R. Alus and G.J. Pappas, editors, *Hybrid Systems: Computation and Control*, number 2993 in LNCS, pages 387–401. Springer-Verlag, Berlin, 2004.
32. J. Hespanha and A.S. Morse. Switching between stabilizing controllers. *Automatica*, 38(11):1905–1917, November 2002.
33. M. Heymann, F. Lin, and G. Meyer. Synthesis and viability of minimally interventive legal controllers for hybrid systems. *Discrete Event Dynamic Systems: Theory and Applications*, 8(2):105–135, June 1998.
34. R. Horowitz and P. Varaiya. Control design of an automated highway system. *Proceedings of the IEEE*, 88(7):913–925, July 2000.
35. B. Hu, X. Xu, P.J. Antsaklis, and A.N. Michel. Robust stabilizing control laws for a class of second order switched systems. *Systems & Control Letters*, 38(3):197–207, 1999.
36. J. Hu, J. Lygeros, and S.S. Sastry. Towards a theory of stochastic hybrid systems. In Nancy Lynch and Bruce H. Krogh, editors, *Hybrid Systems: Computation and Control*, number 1790 in LNCS, pages 160–173. Springer-Verlag, Berlin, 2000.
37. K.H. Johansson, M. Egerstedt, J. Lygeros, and S.S. Sastry. On the regularization of Zeno hybrid automata. *Systems and Control Letters*, 38(3):141–150, 1999.
38. X.D. Koutsoukos, P.J. Antsaklis, J.A. Stiver, and M.D. Lemmon. Supervisory control of hybrid systems. *Proceedings of the IEEE*, 88(7):1026–1049, July 2000.
39. M. Kvasnica, P. Grieder, M. Baotic, and M. Morari. The multi-parametric toolbox (MPT). In R. Alur and G.J. Pappas, editors, *Hybrid Systems Computation and Control*, number 2993 in LNCS, pages 448–462. Springer-Verlag, Berlin, 2004.
40. D. Liberzon and A.S. Morse. Basic problems in stability and design of switched systems. *IEEE Control Systems Magazine*, 19:59–70, October 1999.
41. J. Lygeros, D.N. Godbole, and S.S. Sastry. Verified hybrid controllers for automated vehicles. *IEEE Transactions on Automatic Control*, 43(4):522–539, April 1998.

42. J. Lygeros, K.H. Johansson, S.N. Simić, J. Zhang, and S.S. Sastry. Dynamical properties of hybrid automata. *IEEE Transactions on Automatic Control*, 48(1):2–17, January 2003.
43. J. Lygeros, C.J. Tomlin, and S.S. Sastry. Controllers for reachability specifications for hybrid systems. *Automatica*, 35(3):349–370, March 1999.
44. John Lygeros. On reachability and minimum cost optimal control. *Automatica*, 40(6):917–927, 2004.
45. O. Maler. On optimal and suboptimal control in the presence of adversaries. In *Proceedings of WODES04*, pages 1–12. 2004.
46. Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Springer-Verlag, Berlin, 1992.
47. X. Mao. Stability of stochastic differential equations with Markovian switching. *Stochastic Processes and Applications*, 79:45–67, 1999.
48. D. Mayne and S. Rakovic. Model predictive control of contrained, piecewise affine, discrete time systems. *International Journal of Robust and Nonlinear Control*, 13(3):261–279, 2003.
49. J.L. Menaldi. Stochastic hybrid optimal control models. *Aportaciones Matematicas (Sociedad Matematica Mexicana)*, 16:205–250, 2001.
50. I. Mitchell, A.M. Bayen, and C.J. Tomlin. Validating a Hamilton–Jacobi approximation to hybrid system reachable sets. In M. Di Benedetto and A. Sangiovanni-Vincentelli, editors, *Hybrid Systems: Computation and Control*, number 2034 in LNCS, pages 418–432. Springer-Verlag, Berlin, 2001.
51. D.L. Pepyne and C.G. Cassandras. Optimal control of hybrid systems in manufacturing. *Proceedings of the IEEE*, 88(7):1108–1123, July 2000.
52. B. Piccoli. Necessary conditions for hybrid optimization. In *IEEE Conference on Decision and Control*, pages 410–415, Phoenix, Arizona, December 7–10, 1999.
53. P. J. G. Ramadge and W. M. Wonham. The control of discrete event systems. *Proceedings of the IEEE*, Vol.77(1):81–98, 1989.
54. P. Saint-Pierre. Approximation of viability kernels and capture basins for hybrid systems. In *European Control Conference*, pages 2776–2783, Porto, September 4–7, 2001.
55. A. V. Savkin, E. Skafidas, and R.J. Evans. Robust output feedback stabilizability via controller switching. *Automatica*, 35(1):69–74, 1999.
56. M.S. Shaikh and P.E. Caines. On the optimal control of hybrid systems: Optimal trajectories, switching times and location schedules. In O. Maler and A. Pnueli, editors, *Hybrid Systems: Computation and Control*, number 2623 in LNCS, pages 466–481. Springer-Verlag, Berlin, 2003.
57. M. Song, T.J. Tarn, and N. Xi. Integration of task scheduling, action planning and control in robotic manufacturing systems. *Proceedings of the IEEE*, 88(7):1097–1107, July 2000.
58. Hector J. Sussmann. A maximum principle for hybrid optimal control problems. In *IEEE Conference on Decision and Control*, pages 425–430, Phoenix, Arizona, December 7–10, 1999.
59. W. Thomas. On the synthesis of strategies in infinite games. In Ernst W. Mayr and Claude Puech, editors, *Proceedings of STACS 95*, volume 900 of *LNCS*, pages 1–13. Springer-Verlag, Berlin, 1995.
60. C.J. Tomlin, J. Lygeros, and S.S. Sastry. A game theoretic approach to controller design for hybrid systems. *Proceedings of the IEEE*, 88(7):949–969, July 2000.

61. C.J. Tomlin, G.J. Pappas, and S.S. Sastry. Conflict resolution for air traffic management: A case study in multi-agent hybrid systems. *IEEE Transactions on Automatic Control*, 43(4):509–521, 1998.
62. M. Wicks, P. Peleties, and R. De Carlo. Switched controller synthesis for the quadratic stabilization of a pair of unstable linear systems. *European Journal of Control*, 4:140–147, 1998.
63. H. Wong-Toi. The synthesis of controllers for linear hybrid automata. In *IEEE Conference on Decision and Control*, pages 4607–4613, San Diego, California, December 10–12 1997.
64. X. Xu and P.J. Antsaklis. Stabilization of second order LTI switched systems. *International Journal of Control*, 73(14):1261–1279, September 2000.
65. X. Xu and P.J. Antsaklis. An approach for solving general switched linear quadratic optimal control problems. In *IEEE Conference on Decision and Control*, Orlando, Florida, December 16-18 2001.
66. X. Xu and G. Zhai. On practical stability and stabilization of hybrid and switched systems. In R. Alur and G.J. Pappas, editors, *Hybrid Systems Computation and Control*, number 2993 in LNCS, pages 615–630. Springer-Verlag, Berlin, 2004.
67. C. Yuan and J. Lygeros. Stabilization of a class of stochastic systems with Markovian switching. In *Mathematical Theory of Networks and Systems (MTNS04)*, Leuven, Belgium 2004.
68. C. Yuan and X. Mao. Asymptotic stability in distribution of stochastic differential equations with Markovian switching. *Stochastic Processes and Applications*, 103:277–291, 2003.

Temporal Logic Model Checking

Edmund Clarke,[1] Ansgar Fehnker,[2] Sumit Kumar Jha[1], and Helmut Veith[3]

[1] School of Computer Science, Carnegie Mellon University, PA 15213, Pittsburgh, U.S.A.
{emc,jha+}@cs.cmu.edu
[2] National ICT Australia, University of New South Wales, Australia
ansgar@cse.unsw.edu.au
[3] Institut für Informatik, Technische Universität München, Munich, Germany
veith@in.tum.de

1 Introduction and Overview

Errors in safety-critical systems such as embedded controllers may have drastic consequences and can even endanger human life. It is therefore crucially important to verify the correctness of such systems in a logically precise manner during system design itself. This chapter is an introduction to model checking—an automated and practically successful approach for the formal verification of the correctness of hardware and software systems.

The origins of model checking date back to the early 1980s, when Clarke and Emerson [8] and, independently, Queille and Sifakis [26] introduced a new algorithmic approach for the verification of computer systems. Their approach amounts to checking the satisfaction of a logical specification over a system model which is represented by an annotated directed graph; hence, the term *model checking*. Prior to that, the use of temporal logic for the analysis and specification of computer systems had been advocated by Pnueli [25], and model checking has in fact been employing variants of temporal logic as the predominant specification language ever since. Experiments with early model checkers quickly made clear that the size of the model represents the crucial technical barrier for realizing the full potential of model checking in practically relevant verification tasks. In turn, the *state explosion problem* is the key to appreciating the technical achievements in model checking during the last decades. At the time of this writing, model checking techniques have achieved practical significance for the hardware and software industries, routinely analyzing digital circuit designs and programs, with more than 10^{100} system states in some cases.

The aim of this chapter is to introduce those important lines of research which transformed model checking from a method of primarily theoretical interest into a powerful tool for the analysis of computer hardware and soft-

ware. We shall focus in particular on those subjects which have shaped our thinking about model checking in the verification group of Carnegie Mellon University, most notably symbolic model checking and abstraction. The development of *symbolic model checker* [6, 24] was arguably a turning point in the formal methods field. Employing a combination of binary decision diagrams and fixed-point algorithms, the symbolic model verifier (SMV) became the first model checker to verify models with hundreds of Boolean variables and a tool to benchmark new ideas for more than a decade. Thus, after a brief theoretical introduction into logical foundations of model checking in Section 2, we will describe the methodology behind SMV in Section 3.1; we also cover bounded model checking, a more recent orthogonal symbolic model checking paradigm which is based on SAT solvers. Sections 3.2 and 3.3 finally are devoted to abstraction, the key principle underlying the big advances in software verification during the last few years. The focus in these sections will be on counterexample-guided abstraction refinement as well as predicate abstraction, both of which constitute key features of modern software verification tools.

Most of the material included in this chapter is self-contained, requiring only a general mathematical maturity. However, we explicitly advise the reader that the space restrictions imposed by the handbook format render it impossible to provide either a comprehensive coverage of the subject or even an extended bibliography which gives full credit for all presented concepts impossible. The papers and books cited here should serve mainly as entry points to the literature, with an emphasis on newer papers not yet listed in the standard literature. The most evident omission in this chapter is an extensive treatment of the automata-theoretic approach to model checking [30]; just like temporal logic model checking, the alternative automata-theoretic approach has also produced powerful model checkers such as SPIN. A more comprehensive survey on model checking including extensive citations can be found in [12], more detailed accounts on logical questions in [13, 15], an introduction to SPIN and the automata-theoretic method in [21], and an easily accessible primer on logic and verification in [22].

2 Fundamentals of Model Checking

A model checker is an algorithm which determines whether a system $\mathbf{K}$ satisfies a specification ϕ, formally $\mathbf{K} \models \phi$. In contrast to stochastic methods such as testing, a positive result of the model checker provides a logically precise assertion of system correctness—albeit not in terms of a step-by-step-proof, but by virtue of the construction and correctness of the model checking algorithm. If the system $\mathbf{K}$ is found to violate the specification ϕ, i.e., $\mathbf{K} \not\models \phi$, then most model checkers will compute a diagnostic counterexample $\mathbf{C}$ which helps to localize the source of the error.

The fundamental notion of model checking has been adopted to diverse application areas and formal methods even beyond verification; these areas cannot all be treated in detail within the scope of this chapter. We shall therefore concentrate on the classical model checking framework where the system $\mathbf{K}$ is given as a *Kripke structure*, and the specification ϕ is a *temporal logic formula*.

Kripke structures

The notion of a *state* is at the center of model checking. A state is a momentary description of a system at a given point in time, similar to a point in a physical phase space. When a system has only a finite number of possible states, we speak of a finite state system.

A Kripke structure describes the dynamics of a finite state system by a finite directed graph whose vertices denote the states and whose edges denote transitions between the states. The states are labelled by atomic propositions which denote the properties of each state. For example, the states of the Kripke structure can be taken to describe the different states of a microprocessor, and the labels describe the values of the registers associated with a state. In the simple examples used in this chapter, the atomic propositions will typically just have the form of Boolean variables; in practice, atomic formulas often describe properties of the internal variables of the real-life system we model, e.g., "`counter == 5`" or "`x == -1`".

Formally, a Kripke structure is a tuple $\mathbf{K} = (S, S_0, R, L, \mathrm{AP})$ where S is a finite set of states, $S_0 \subseteq S$ is the set of initial states, $R \subseteq S \times S$ is the transition relation, AP is the set of atomic properties, and $L : S \to 2^{\mathrm{AP}}$ is the labelling function. R is required to be total, i.e., for each $s \in S$ there exists $t \in S$ such that $(s, t) \in R$. When the context is understood, S_0 and AP are often omitted. For simplicity, we will assume in this chapter that $S_0 = \{s_0\}$ contains a single initial state.

A path π is an *infinite* sequence of states $\pi = p_0, p_1, \ldots$ such that for all i, $(p_i, p_{i+1}) \in R$. For $i \geq 0$, we define $\pi^i = p_i, p_{i+1}, \ldots$ to be the path starting at p_i, and $\pi[i] = p_i$. Note that the totality of R guarantees that all finite paths can be extended infinitely.

Example 1. The Kripke structure $\mathbf{M}$ in Fig. 1 shows a simple example of mutual exclusion between two processes A and B. Label C_X denotes that process X is in the critical state, and label T_X denotes that X is trying to enter its critical section. By visual inspection, the reader will easily verify that mutual exclusion is guaranteed, i.e., that no state can be reached where both C_A and C_b hold. Note that this example of mutual exclusion is very simplified: no fairness guarantees are given and a process may stay in the critical section forever.

Several remarks about the use of Kripke structures in model checking are in place here:

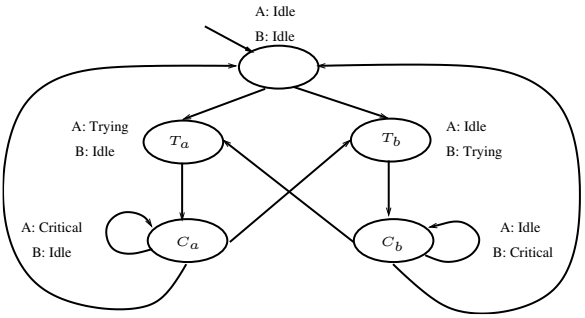


Fig. 1. A Kripke structure modelling a trivial mutual exclusion protocol

(*a*) Kripke structures are a versatile *mathematical model* we consider in model checking. Practical model checkers use specific programming languages and not Kripke structures as their *input language*. The complexity issues arising from direct compilation of a program into a Kripke structure constitute the central algorithmic challenge of model checking (the "state explosion problem") and will be addressed in Section 3.1.

(*b*) Kripke structures are in general *non-deterministic*, i.e., from a given state s there will be more than one outgoing transition. Non-determinism is a natural way to describe the effects of external input to a system. In Section 3.2 we will see that non-determinism also arises when we approximate the behavior of large systems by relatively small Kripke structures.

(*c*) Kripke structures are closely related to finite automata, finite transition systems, Moore machines, process algebraic expressions and similar concepts whose differences are to a large extent rooted in pragmatic aspects and tradition. In principle, every finite state system can be represented by a Kripke structure. In Section 3.3 we will investigate extensions of model checking to deal with infinite state systems.

Temporal logic

Given the atomic propositions of the Kripke structure, we can use Boolean logic to describe compound properties of *single* states. For example, we write $\mathbf{K}, s \models f \wedge \neg g$ to denote that state s of system $\mathbf{K}$ has label f, but not label g. However, simple Boolean logic does not account for the temporal dynamics of systems: In Boolean logic we cannot express properties such as "f is an invariant", "f always precludes g", or "f will persist up to the time when g occurs". This is where the temporal logics LTL, CTL and CTL* come into play.

We will first describe the linear time logic (LTL) which is defined over paths of the system. LTL extends Boolean logic by two operators U and X. Given a path $\pi = p_0, p_1, \ldots$, the Boolean and temporal operators have the following recursive semantics:

$\mathbf{K}, \pi \models f$ iff $f \in L(\pi[0])$ where $f \in \mathrm{AP}$
$\mathbf{K}, \pi \models \phi \wedge \psi$ iff $\mathbf{K}, \pi \models \phi$ and $\mathbf{K}, \pi \models \psi$
$\mathbf{K}, \pi \models \neg\phi$ iff $\mathbf{K}, \pi \not\models \phi$

$\mathbf{K}, \pi \models \mathsf{X}\phi$ iff $\mathbf{K}, \pi^1 \models \phi$
" ϕ is true in the next state"

$\mathbf{K}, \pi \models \phi \mathsf{U} \psi$ iff there is an $i \geq 0$ such that $\mathbf{K}, \pi^i \models \psi$
and for all j with $0 \leq j < i$ we have $\mathbf{K}, \pi^j \models \phi$
" ϕ is true until ψ becomes true"

$\mathbf{K} \models \phi$ iff for all paths π starting in s_0, we have $\mathbf{K}, \pi \models \phi$.

Disjunction $\phi \vee \psi$ and implication $\phi \rightarrow \psi$ are as usual defined by $\neg(\neg\phi \wedge \neg\psi)$ and $\neg(\phi \wedge \neg\psi)$, respectively. Moreover, $\mathsf{F}\psi$) and $\neg(\phi \wedge \neg\psi)$, respectively. Moreover, $\mathsf{F}\phi$ is defined $\mathbf{true}\mathsf{U}\phi$, and $\mathsf{G}\phi$ is defined $\neg\mathsf{F}\neg\phi$. With these definitions, $\mathsf{F}\phi$ intuitively means *"ϕ will be true at some time in the future"* and $\mathsf{G}\phi$ means *"ϕ will always be true in the future."*

Note that LTL specifications describe properties of paths; an LTL specification holds true on a Kripke structure, if it holds true on *every path* starting at the initial state s_0. Thus, an LTL specification contains an implicit universal quantification over all paths starting at s_0.

We will now introduce the computational tree logics CTL* and CTL which enable us to quantify over paths explicitly. CTL* extends LTL by an operator A as follows:

$\mathbf{K}, \pi \models \mathsf{A}\phi$ iff for all paths σ starting in state $\pi[0]$, we have $\mathbf{K}, \sigma \models \phi$.

Existential path quantification $\mathsf{E}\phi$ is defined as an abbreviation for $\neg\mathsf{A}\neg\phi$. The important specification logic CTL is the syntactic fragment of CTL* which uses the LTL operators and the CTL operators only pairwise, i.e., CTL contains exactly the following temporal operators: $\mathsf{AX}, \mathsf{EX}, \mathsf{AU}, \mathsf{EU}, \mathsf{AF}, \mathsf{EF}, \mathsf{AG}, \mathsf{EG}$. For example, the CTL* formula $\mathsf{AXX}f$ is not in CTL, since the second occurrence of X is not preceded by E or A. Finally, ACTL (ACTL*) is the fragment of CTL (CTL*) where negation is restricted to atomic formulas, and only the path quantifier A is allowed.

CTL and LTL specifications

Since CTL specifications can quantify repeatedly over paths, CTL and CTL* are examples of *branching time logics*, while LTL is a *linear time logic*. It can be shown that the expressive power of CTL and LTL is not comparable, and both are strictly contained in CTL*. The algorithms and paradigms for CTL and LTL model checking are sufficiently different so as to provoke a controversial discussion in the literature as to which is preferable, branching

time or linear time logic. Practical applications employ variants of either CTL or LTL.

Example 2. In the example Kripke structure $\mathbf{M}$ used previously, the mutual exclusion property is specified in CTL as $\mathsf{AG}\neg(C_a \wedge C_b)$. The *liveness property* that each of the processes enters its critical region infinitely often is given by the CTL formula $\mathsf{AG}(\mathsf{AF}(C_a) \wedge \mathsf{AF}(C_b))$. The two properties are also expressible in LTL by $\mathsf{G}\neg(C_a \wedge C_b)$ and $\mathsf{GF}(C_a) \wedge \mathsf{GF}(C_b)$, respectively.

To shed more light on the difference between CTL and LTL, let us consider the equivalence relation between Kripke structures induced by CTL and LTL specifications, i.e., we say that two Kripke structures are equivalent if there is no CTL (or LTL) specification which holds true for one, but not for the other. For LTL, this equivalence relation is known as *trace equivalence*, i.e., two Kripke structures are trace equivalent iff they have the same paths from the initial state. For CTL, in contrast, the equivalence relation is *bisimulation*, a central notion in process algebra which we describe in more detail below.

Bisimulation and simulation

Bisimulation can be defined by a combinatorial two-player game [28] where one player (the *spoiler*) attempts to show that two Kripke structures $\mathbf{A}$ and $\mathbf{B}$ are different, and the second player (the *duplicator*) attempts to show that the structures are equivalent. The game starts with two pebbles placed on the initial states of the two structures. Each round of the game proceeds as follows: (i) If the pebbles are located at states with different labels, the spoiler wins, and the game terminates. (ii) The spoiler chooses one Kripke structure, and moves the pebble along an edge in this Kripke structure. (iii) In the other Kripke structure, the duplicator moves the other pebble, and the game continues at step (i). The Kripke structures are bisimilar if the spoiler does not have a winning strategy.

If the spoiler has a winning strategy, i.e., if the Kripke structures are not bisimilar, then the strategy can already be expressed by a CTL formula which uses the temporal operators AX and EX and distinguishes $\mathbf{A}$ from $\mathbf{B}$. Conversely, a distinguishing specification gives rise to a winning strategy for the spoiler.

Closely related to bisimulation is the notion of *simulation*, which orders Kripke structures with respect to their behaviors. For two Kripke structures, $\mathbf{A}$ and $\mathbf{B}$, simulation $\mathbf{A} \preceq \mathbf{B}$ can be defined by a similar two-player game as above, with the following restriction: the spoiler always plays on Kripke structure $\mathbf{A}$, and the duplicator always plays on Kripke structure $\mathbf{B}$. If the spoiler does not have a winning strategy, then $\mathbf{A} \preceq \mathbf{B}$ holds true. Intuitively, in this case, $\mathbf{B}$ has more behavior than $\mathbf{A}$, as the duplicator can duplicate every move on $\mathbf{B}$ which the spoiler does on $\mathbf{A}$. Simulation has the important property that it preserves ACTL* specifications: If $\mathbf{A} \preceq \mathbf{B}$ and $\mathbf{B} \models \phi$ for an

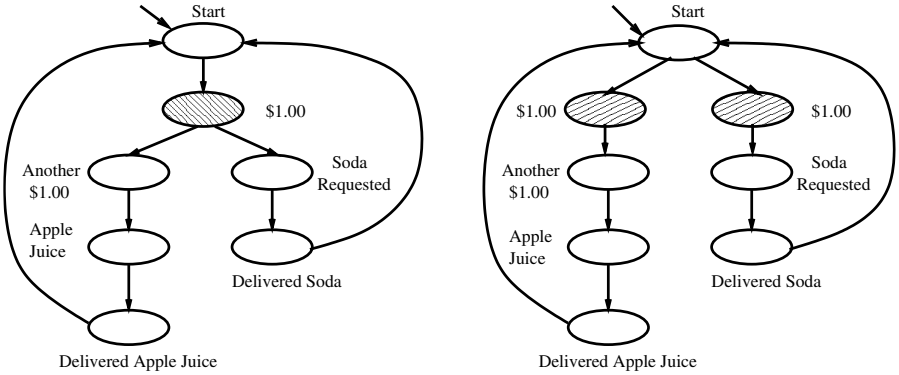


Fig. 2. Simulation versus bisimulation

ACTL* specification ϕ, then $\mathbf{A} \models \phi$. We will see that this relationship has crucial algorithmic applications in model checking.

It is easy to see that bisimulation equivalence implies trace equivalence: suppose the Kripke structure $\mathbf{A}$ has a path π_1 not contained in $\mathbf{B}$. Then there exists a finite prefix π_1' of π_1 which is not present in $\mathbf{B}$. It is now easy to construct a CTL specification of the from $\mathsf{EX}(\ldots \wedge \mathsf{EX}(\ldots \wedge \mathsf{EX}(\ldots)))$ which stipulates the existence of π_1'. Example 3 demonstrates a classical case where we have trace equivalence, but not bisimulation equivalence.

Example 3. Fig. 2 demonstrates a case where two Kripke structures are trace equivalent but not bisimilar. Both describe a simplified vending machine for soda (at \$1 per serving) and apple juice (at \$2). In the right machine, money is inserted in separate slots, and thus, the first dollar determines the purchase. It is easy to see that both structures have the same traces, i.e., they are trace equivalent, but not bisimilar. This is also reflected in the bisimulation game: When the spoiler moves the pebble to the shaded state in the left structure he keeps a future choice between juice and soda, but in the right structure, the duplicator has to decide on his tastes at this moment. It is easy to see that both structures have the same traces, i.e., they are trace-equivalent, but not bisimilar. This is also reflected in the bisimulation game: When the spoiler moves the pebble to the shaded state in the left structure, he keeps a future choice between juice and soda, but the duplicator has to decide on his tastes at this moment in the right structure.

Similar situations occur, for example, in computer security when instead of coins we enter passwords which enable us to call operating system functions. In such situations, the difference between bisimulation and trace equivalence may become crucial.

Principle algorithmic aspects of model checking

In contrast to many other applications of logic in computer science, most questions related to temporal logics and model checking are decidable, and indeed

have often reasonable complexity. Quite surprisingly, the problem of deciding $\mathbf{K} \models \phi$ can be solved in linear time $O(|\phi| \times (|S| + |R|))$ for CTL. This explicit CTL model checking algorithm proceeds bottom-up on the formula structure which yields the factor $|\phi|$ above. For each subformula ϕ', the algorithm labels the states of $\mathbf{K}$ which satisfy ϕ' in time $O(|S| + |R|)$. For LTL and CTL*, the model checking problem is complete for the complexity class PSPACE. Deciding the *validity* of specifications, i.e., determining whether a specification is always true (independent of the Kripke structure) is EXPTIME-complete for CTL, PSPACE-complete for LTL and 2EXPTIME-complete for CTL*. In practice, however, the complexity bounds obtained from these general abstract considerations are usually not sufficient to verify industrially relevant systems. This question will be addressed in the next section.

3 State Explosion and Efficient Verification Methods

In most practical applications of model checking, the size of the state space is too large by several magnitudes as to allow naive verification algorithms which compile the input into a Kripke structure, and perform the model checking algorithm thereafter. Even an extremely simple system containing three 32-bit integer variables has a theoretical state space of $(2^{32})^3$. A quick calculation shows that for a terahertz processor which can evaluate one state per system cycle, it will take around 10^9 years to enumerate all states. Since the state space is in general exponential in the memory a program uses, it is entirely unrealistic to perform model checking on an explicit Kripke structure. This principal problem, commonly referred to as the "state explosion problem," is the core issue in most scientific research in model checking. The practical success of a model checking technique depends most crucially on its ability to alleviate the state explosion. We distinguish several principal ways to address state explosion.

- **Symbolic verification:** In this approach, which made model checking a practical technique, the transition relation of the Kripke structure is encoded in a Boolean formalism (either plain Boolean formulas or specific data structures such as ordered binary decision diagrams (OBDDs) [5]), thereby obtaining a potentially exponential compression factor. Specific "symbolic" model checking algorithms are devised to operate on such system representations. We will describe symbolic model checking methods for CTL and LTL below.
- **State space exploration:** Alternatively to symbolic representation, a model checking algorithm may also attempt to explore the state space for specification violations on the fly, i.e., by a systematic depth-first search starting from the initial state. This method is based on the insight that LTL specifications can be transformed into trace equivalent *Büchi automata* [12] which monitor the state space exploration. While the size of

the state space sets a principal limit to this approach, it is successful in *finding errors*, in particular in combination with abstraction and reduction methods (described next).

- **Abstraction and reduction methods:** In this category we subsume more aggressive methods which are typically orthogonal to both symbolic and exploration techniques. Their common characteristic is their attempt to restrict the state space by semantic considerations, i.e., properties known about the system or derived from its description. Typical examples of such methods include *abstraction* [11], where system states are partitioned into equivalence classes, *partial order reduction* [18], which curbs the state space explosion incurred by concurrency, or *symmetry reductions* [9, 16], which employ the natural symmetry between repeated system components. In Section 3.2 we shall describe abstraction and counterexample-based abstraction refinement in more detail.

3.1 Symbolic model checking

Recall from above that the characteristic step in symbolic model checking is to represent the transition relation R of a Kripke structure $\mathsf{K} = (S, S_0, R, L, \mathrm{AP})$ in terms of a Boolean function f_R in such a way that every state $s \in S$ is described by a unique Boolean vector $\bar{s}$, and $f_R(\bar{s}, \bar{t}) = \textsf{true}$ iff $(s, t) \in R$. Since for natural binary encoding the size of the Boolean vector is logarithmic in $|S|$, the size of f_R may—in principle—also be polynomial or even linear in $\log |S|$. While there is no mathematical guarantee for this compression to occur (in fact, information theoretic counting arguments easily show that such a compression is very rare), practical systems tend to have many regularities, and often allow significant compression.

Note that in the computation of f_R, the model checker does not have access to the Kripke structure K (which is too large by assumption), but only to the input program. In this setting, choosing binary representations $\bar{s}$ for states s is usually a very natural step, since each system state s describes a program state at a given time, and thus corresponds to specific values of the program variables; these program variables themselves have natural binary representations which the model checker can reuse. In fact, a close correspondence between symbolic variables and program variables is often advantageous: knowledge about the semantic relationship between symbolic variables can facilitate both compression and abstraction.

CTL verification: verification by fixed point computation

Recall that in the specification logic CTL, every LTL operator is immediately preceded by either E or A. Since the semantical definitions of $\mathsf{K}, \pi \models \mathsf{A}\phi$ and $\mathsf{K}, \pi \models \mathsf{E}\phi$ depend only *on the first state* $\pi[0]$ of path π, formulas with a leading A or E are called state formulas. A model checking algorithm can associate each CTL state formula ϕ with the set of states $[[\phi]]$ where ϕ holds true. For

CTL formulas ϕ and Kripke structures $\mathbf{K}$, the set $[[\phi]]$ can be computed by a fixed-point algorithm which can be implemented symbolically.

To illustrate the principles of symbolic model checking for CTL, let us consider the specification $\mathsf{EF}\phi$ ("a state with property ϕ is reachable"). Given the set $[[\phi]]$ of states where ϕ holds true, $[[\mathsf{EF}\phi]]$ is inductively defined as follows:

- If $s \in [[\phi]]$, then $s \in [[\mathsf{EF}\phi]]$.
- If $s \in [[\mathsf{EF}\phi]]$ and $R(t, s)$, then $t \in [[\mathsf{EF}\phi]]$.
- Nothing else is in $[[\mathsf{EF}\phi]]$.

This gives rise to the fixed-point characterization

$$\mathsf{EF}\phi \equiv \mu Y.\phi \vee \mathsf{EX}\ Y$$

where $\mu Y.f(Y)$ denotes the least fixed-point of formula $f(Y)$. The fixed-point extension of temporal logic is called the μ-calculus, and has been studied extensively, see [12] for detailed definitions and references. From the μ-calculus characterization of $\mathsf{EF}\phi$ we can derive the following fixed-point algorithm to compute $[[\mathsf{EF}\phi]]$.

$Y := \emptyset$
repeat
  $Y' := Y$;
  $Y := [[\phi]] \cup \mathbf{pre}(Y)$;
until $Y = Y'$

where $\mathbf{pre}(Y)$ denotes the *pre-image* operator

$$\mathbf{pre}(Y) := \{s \mid \exists t.(s, t) \in R \wedge t \in Y\}.$$

A closer study shows that all CTL formulas can be expressed using fixed points, propositional logic, and the temporal operator EX, i.e., pre-image computation. It remains to be shown as to how the fixed-point algorithms can be implemented symbolically.

The crucial idea is to represent not only R by f_R, but also sets of states by Boolean functions. A set Y of states is represented by its characteristic Boolean function $Y(\bar{z}) := \bigvee_{s \in Y} \bar{z} \equiv \bar{s}$ which is **true** iff $\bar{z}$ is the binary representation of a state in Y. For two sets Y and Z, its union $Y \cup Z$ is represented by $Y(\bar{z}) \vee Z(\bar{z})$, and similarly for other set operations. Pre-image computation can also be expressed easily in this framework:

$$\mathbf{pre}(Y(\bar{s})) = \exists \bar{t}.f_R(\bar{s}, \bar{t}) \wedge Y(\bar{t}).$$

Since Boolean quantification can be eliminated, the result of $\mathbf{pre}(Y(\bar{s}))$ is again a Boolean function. We conclude that all operations in the fixed-point algorithm can be computed symbolically.

For a practical implementation, it is necessary to have a data structure for Boolean functions which (i) facilitates good compression capabilities, but

(ii) makes it easy to recognize that a fixed point has been reached. Ordinary Boolean functions have the disadvantage that deciding the termination condition $Y(\bar{z}) \equiv Y'(\bar{z})$ of the fixed-point algorithm is coNP-complete, and thus a computationally hard problem. A successful trade-off is achieved by *OBDDs* [5]. OBDDs are compact (although somewhat less so than Boolean functions), and, importantly, they have canonical representations which makes it easy to decide $Y(\bar{z}) \equiv Y'(\bar{z})$ efficiently.

OBDDs

Let AP be the set of propositional variables, and $<$ a linear order on AP[1]. An OBDD O over AP is an acyclic graph (V, E) whose non-terminal vertices (*nodes*) are labelled by variables from A, and whose edges and terminal nodes are labelled by 0, 1. Each non-terminal node v has out-degree 2, such that one of its outgoing edges is labelled 0 (the *low edge* or *else-edge*), and the other is labelled 1 (the *high edge* or *then-edge*). If v has label a_i and the successors of v are labelled a_j, a_k, then $a_i < a_j$ and $a_i < a_k$. In other words, for each path, the sequence of labels along the path is strictly increasing with respect to $<$.

Each OBDD node v represents a Boolean function O_v. The terminal nodes of O represent the constant functions given by their labels. A non-terminal node v with label a_i whose successors at the high and low edges are u and w, respectively, defines the function $\mathsf{O}_v := (a_i \wedge \mathsf{O}_u) \vee (\neg a_i \wedge \mathsf{O}_w)$. For every variable order $<$ and Boolean function f there exists a *canonical minimal* OBDD O over AP which represents the Boolean function f. Given any OBDD for f which respects $<$, the canonical OBDD O can be computed in polynomial time. Thus, with OBDDs, set operations including equality testing of sets can be efficiently complemented. Pre-image computation, however, is much harder; in fact, it is one of the major bottlenecks in verification. Another problem with OBDDs is the prevalence of state explosion: for certain functions, the size of the minimal OBDD may be exponential in AP, and moreover the size crucially depends on the variable order $<$. A simple example of a binary decision diagram (BDD) is given in Fig. 3.

LTL verification: bounded model checking.

Bounded model checking [3] is a new method which leverages the surprising power of recent SAT solvers, i.e., algorithms which on input of a Boolean formula search for a satisfying assignment[2]. Recall that LTL specifications express properties which have to hold over *all* paths; consequently, a counterexample for an LTL property is given by a single path which violates the

[1] In this section we assume for simplicity that all states in the Kripke structure are uniquely identified by their labels, i.e., that the labelling function L is injective.

[2] Note that Boolean satisfiability is a prototype NP-complete problem, and thus we cannot expect a SAT solver to scale polynomially for all inputs. However, state-of-the-art SAT-solvers are remarkably successful at a large portion of those formulas which occur in practical verification tasks.

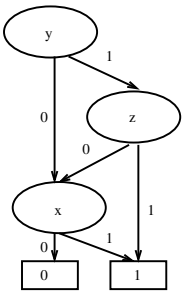


Fig. 3. A BDD for $(y \wedge z) \vee (y \wedge \neg z \wedge x)$. Note that the size of the diagram in this case is linear in the number of Boolean variables.

specification. Consider for example a specification of the form $\mathsf{G}b$, i.e., "always b". Then a counterexample for $\mathsf{G}b$ is a path where at some point $\neg b$ holds. Suppose that for a state s, $b(\bar{s})$ is the Boolean formula expressing that b holds at state s. Then the formula

$$f_R(\bar{s}_0, \bar{s}_1) \wedge f_R(\bar{s}_1, \bar{s}_2) \wedge \ \cdots \ \wedge f_R(\bar{s}_{k-1}, \bar{s}_k) \wedge \bigvee_{0 \le i \le k} \neg b(\bar{s}_i)$$

is satisfiable if and only if a counterexample of size $\leq k$ exists; consequently, a SAT solver can be used to decide the existence of a counterexample. Similarly, it can be shown that for any fixed counterexample length k, any LTL specification can be translated into a SAT instance. Moreover, the satisfying assignment computed by the SAT solver can be used easily to compute the counterexample.

Bounded model checking has been used successfully for both hardware and software, and has outperformed BDD-based methods on various examples. The evident drawback of bounded model checking, however, is its inherent incompleteness: unless the bound k is chosen to be significantly larger than $|S|$, bounded model checking provides no assertion about the total absence of counterexamples. Consequently, bounded model checking is mainly considered a method to find errors rather than a complete verification tool.

3.2 Counterexample-guided abstraction refinement

Abstraction reduces the state space by removing irrelevant features of a Kripke structure. Given a Kripke structure $\mathbf{K}$, an abstraction is a Kripke structure $\widehat{\mathbf{K}}$ such that $\widehat{\mathbf{K}}$ is significantly smaller than $\mathbf{K}$, and $\widehat{\mathbf{K}}$ *preserves* a useful class of specifications for $\mathbf{K}$. Consequently, the expensive task of model checking $\mathbf{K}$ can be reduced to the more feasible task of model checking $\widehat{\mathbf{K}}$. We know from above that in order to preserve all CTL specifications, $\mathbf{K}$ and $\widehat{\mathbf{K}}$ must be bisimilar. But bisimilarity, by its very definition, expresses that $\mathbf{K}$ and $\widehat{\mathbf{K}}$ are behaviorally equivalent. Consequently, $\widehat{\mathbf{K}}$ still models a lot of irrelevant behavior and will therefore be quite large in general.

A more practical approach is to employ the fact explained in Section 2 that simulation preserves ACTL* formulas, i.e., $\mathbf{A} \preceq \mathbf{B}$ and $\mathbf{B} \models \phi$ imply $\mathbf{A} \models \phi$. Consequently, for an abstract system $\widehat{\mathbf{K}}$ where $\mathbf{K} \preceq \widehat{\mathbf{K}}$ holds, a successful run of the model checker over $\widehat{\mathbf{K}}$ implies correctness over the original Kripke structure $\mathbf{K}$, *without model checking* $\mathbf{K}$. The converse implication, however, will not hold in general: an ACTL* property which is false in $\widehat{\mathbf{K}}$ may still be true in $\mathbf{K}$. In this case, the abstract counterexample obtained over $\widehat{\mathbf{K}}$ cannot be reconstructed for the concrete Kripke structure $\mathbf{K}$, and is called a *spurious counterexample* [10], or a false negative.

An important instance of simulation-based abstraction is *existential abstraction* [11, 14] where the abstract states are essentially equivalence classes of concrete states; a transition between two abstract states holds if there was a transition between any two concrete member states in the corresponding equivalence classes. Formally, an abstraction function h is a surjection $h : S \to \widehat{S}$ where $\widehat{S}$ is the set of *abstract states*. The surjection h induces an equivalence relation $\equiv$ on the state space S where $d \equiv e$ iff $h(d) = h(e)$. The abstract Kripke structure $\widehat{\mathbf{K}} = (\widehat{S}, \widehat{S_0}, \widehat{R}, \widehat{L}, \mathrm{AP})$ derived from h is defined as follows:

$$\begin{aligned}
\widehat{S_0} &= \{\widehat{d} \mid \exists d \in S_0 \,.\, h(d) = \widehat{d}\} \\
\widehat{R} &= \{(\widehat{d_1}, \widehat{d_2}) \mid \exists d_1, d_2 \in S \,.\, h(d_1) = \widehat{d_1} \wedge h(d_2) = \widehat{d_2} \wedge R(d_1, d_2)\} \\
\widehat{L}(\widehat{d}) &= \bigcup_{h(d)=\widehat{d}} L(d)
\end{aligned}$$

We also write $\widehat{\mathbf{K}} = \mathbf{K}/\!\equiv$ to express the dependence of $\widehat{\mathbf{K}}$ on $\equiv$. An atomic proposition $f \in \mathrm{AP}$ *respects* an abstraction function h if for all d and d' in the domain S, $(d \equiv d') \Rightarrow (d \models f \Leftrightarrow d' \models f)$. When the specifications contain only atomic propositions respecting h, we can without loss of generality assume that in both $\mathbf{K}$ and $\widehat{\mathbf{K}}$, AP is restricted to the propositions occurring in the specifications. Then existential abstraction indeed guarantees $\mathbf{K} \preceq \widehat{\mathbf{K}}$ as intended.

However, determining a good abstraction function h is a difficult task: If $\widehat{K}$ is too large, then verification remains infeasible. If, on the other hand, $\widehat{K}$ is too small, then spurious counterexamples are likely to occur, as illustrated by the next example. The example also illustrates that abstraction typically introduces non-determinism.

Example 4. Fig. 4 shows two abstract Kripke structures $\widehat{\mathbf{M}_1}$ and $\widehat{\mathbf{M}_2}$ obtained from the original Kripke structure $\mathbf{M}$ by two different equivalence relations $\equiv_1$ and $\equiv_2$. $\mathbf{M}$ describes a simplified bus arbiter that controls access on two buses. The arbiter chooses one of the two buses for any request before giving a grant. It asserts A, B and C to suggest the slave to use bus 1; otherwise, it asserts D, E and F to suggest the slave to use bus 2 and then gives a grant. The slave is supposed to probe which of the lines have been asserted

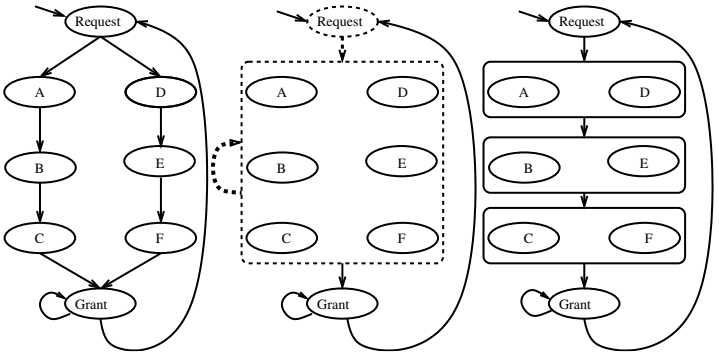


Fig. 4. The Kripke structure $\mathbf{M}$ on the left is existentially abstracted in two ways, yielding abstract Kripke structures $\widehat{\mathbf{M}}_1$ (center figure) and $\widehat{\mathbf{M}}_2$ (right figure). Note that $\widehat{\mathbf{M}}_1$ contains an infinite path which never reaches the state labelled Grant; being unique to the abstract model, this path is called a spurious path.

when it receives the grant and use the appropriate bus. Let us consider the specification $\mathsf{AG}(\mathit{request} \rightarrow \mathsf{AF}\mathit{grant})$ which says that every request is finally granted. By manual inspection we see that the specification holds true for $\mathbf{M}$. The first abstraction $\widehat{\mathbf{M}}_1$ of the arbiter, however, is too coarse, and does not allow us to prove correctness of the specification: we get a counterexample involving a self-loop, as indicated by the dashed lines in the figure. A finer abstraction $\widehat{\mathbf{M}}_2$ passes the specified property and hence, the property is also true for our original Kripke structure $\mathbf{M}$. Note that in both cases $\mathbf{M} \preceq \widehat{\mathbf{M}}_1$ and $\mathbf{M} \preceq \widehat{\mathbf{M}}_2$, i.e., all universal properties on the abstract model are preserved. In the case of $\widehat{\mathbf{M}}_1$, however, preservation does not help, since $\widehat{\mathbf{M}}_2$ exhibits too much information loss for the specification to hold.

Counterexample-guided abstraction refinement (CEGAR) is a natural approach which resolves this situation by using an adaptive algorithm which starts with a coarse abstraction and gradually improves the abstraction function by analyzing spurious counterexamples. CEGAR-style approaches have been investigated by several researchers beginning with the *localization reduction* of Kurshan [23] where the model is abstracted/refined by removing/adding variables from the system description. The first systematic account of CEGAR for CTL model checking was given in [10]. We describe the CEGAR loop using the equivalence relation $\equiv$ induced by h in Fig. 5.

In the CEGAR loop, the abstraction is refined until the property is either verified or disproved by a non-spurious counterexample. Note that the CEGAR loop involves two crucial steps in addition to model checking: the computation of the initial relation $\equiv$, and the computation of the refined abstraction. The initial abstraction is usually obtained by static analysis of the input program (cf. also Section 3.3), and the refinement is achieved by projecting $\widehat{\mathbf{C}}$ back onto $\mathbf{K}$, determining where the spurious behavior occurs, and

Counterexample Guided Abstraction Refinement

$R_{\equiv}$:= initial state equivalence;
result := empty;
repeat
  $\widehat{\mathsf{K}} := \mathsf{K}/R_{\equiv}$
  call model checker for $\widehat{\mathsf{K}} \models \phi$
  if $\widehat{\mathsf{K}} \models \phi$
    then result := "specification true";
    else compute counterexample $\widehat{\mathsf{C}}$;
      if $\widehat{\mathsf{C}}$ is spurious
        then $R_{\equiv}$:= **refine**$(\mathsf{K}, \widehat{\mathsf{C}}, R_{\equiv})$;
        else result := $\widehat{\mathsf{C}}$;
until result not empty;

Fig. 5. General scheme for CEGAR. For better readability, the relation $\equiv$ is written $R_{\equiv}$ in the program text.

locally refining $\equiv$ to eliminate $\widehat{\mathsf{C}}$. Since $\widehat{\mathsf{C}}$ typically is much smaller than $\widehat{\mathsf{K}}$, the projection of $\widehat{\mathsf{C}}$ back onto K involves only a small portion of the state space of K, and is therefore feasible in many practical cases. CEGAR frameworks have become a widely used paradigm in verification, and are routinely used for both hardware and software.

3.3 Model checking for infinite state systems

Model checking was originally designed for the verification of finite state systems. Although the first practically useful applications of model checking were oriented towards hardware verification, where the finite state restriction comes naturally, the method was conceived of as an approach to software verification as well. The early papers on model checking clearly drew their motivations from the software area, focusing in particular on concurrency properties to be verified over the synchronization skeleton of a program, i.e., a finite abstract model which preserves the relevant behavior for interprocess communication. It is still the case that abstraction is one of the key methods to be used for software verification; in fact, model checking and abstract interpretation [14] share many common techniques which deserve further exploration. In this section, we will concentrate on predicate abstraction [19], a particularly important abstraction method which underlies the recent advancements in software verification exemplified by tools such as BLAST, SLAM and MAGIC [2,7,20]. We present a variant of predicate abstraction as it is used in MAGIC.

Predicate abstraction

For the analysis of software, the simplest abstraction which arguably represents the program behavior in a meaningful way is the control flow graph

(CFG) which can be viewed as a Kripke structure where the states are program counter positions, and the transitions denote non-deterministic changes of the control flow. Note that the CFG can be seen as an existential abstract model where the abstraction function h abstracts away everything except the program counter information. It is evident that for many properties of interest, the CFG does not contain sufficient information for verification.

The technique of predicate abstraction [19] is based on the observation that what often counts in the analysis of a program is not so much the actual values of the variables, but rather their relation to each other. Important relations between variables are expressed, for example, in the control conditions which occur in if-statements and in loop headers. Thus, instead of keeping 64 bits for two integer variables x, y in our global state, we may have a single bit which keeps the truth value of the *predicate* $x > y$ if this is the property of interest. In predicate abstraction, we define k such predicates, and extend the CFG by the different evaluations of the predicates. The state space of the new system is 2^k times the size of the CFG, and by choosing the number k of predicates, we can obtain a trade-off between preciseness and state explosion. Thus, in our model, each state of the extended CFG can be described by a formula Ψ of the form

$$(\text{ProgramCounter} = \text{i}) \ \wedge \bigwedge_{1 \le i \le k} \psi_i,$$

where the ψ_i are predicates or negated predicates ranging over the variables of the program. Such a formula Ψ can be identified with an abstract state representing all concrete states (i.e., memory contents) which satisfy Ψ. However, the tricky part in predicate abstraction is to define the transition relation: Suppose that we have a transition in the CFG between program counter positions i and j through a simple statement `statement`, and 2^k abstract states each for i and j, i.e., $\Psi_{i,1}, \ldots, \Psi_{i,2^k}$ and $\Psi_{j,1}, \ldots, \Psi_{j,2^k}$. Potentially, the single transition in the CFG gives rise to up to $(2^k)^2$ transitions in the extended CFG. We actually include a transition from $\Psi_{i,a}$ to $\Psi_{j,b}$ if the weakest precondition required for $\Psi_{j,b}$ to hold after execution of `statement` *is consistent* with $\Psi_{i,a}$, i.e., if

$$\Psi_{i,a} \wedge \text{WP}[\texttt{statement}, \Psi_{j,b}]$$

is logically satisfiable. Deciding satisfiability is in general a hard question, and is often delegated to an automated theorem prover or a decision procedure.

It is not hard to prove that the model obtained by predicate abstraction fits into the simulation-based approach to abstraction; in fact, it is not even necessary for the theorem prover to always terminate. When a theorem prover does not produce a definite answer in due time, we can overapproximate the result by pretending that the theorem prover asserted consistency. It can be shown that the resulting model is still a sound abstract model in this case; if this happens too often, however, the quality of the model will deteriorate towards a very coarse model with extensive spurious behavior. Importantly,

predicate abstraction provides a natural and clean interface between model checking and theorem proving, playing to the strengths of both methods. Predicate abstraction is very successful when verifying control-intensive software such as device drivers or many embedded programs. For complicated data structures, in particular dynamic data structures, predicate abstraction is potentially applicable, but the logics and corresponding decision procedures are in most cases still beyond the state of the art.

Other approaches to infinite systems

Let us finally discuss the question of verification of infinite systems on a broader scale. Precisely speaking, most of the systems we consider are not infinite, but rather parameterized: they are described by a finite program text, and the actual size of the state space depends on a parameter (e.g., the memory size) which is not known in advance, and is often assumed to be arbitrarily large. Besides memory size, other sources of unbounded behavior include recursion depth, the preciseness of floating-point operations, dynamic thread creation, protocols with unlimited number of participants and hybrid systems where parts of the system or the environment are modeled by differential equations as in control theory. The different flavors of infinity we encounter in applications clearly need to be matched by a correspondingly rich set of tools, each tailored for a specific source of infinity. However, in contrast to finite state verification, we cannot realistically expect to find methods which apply uniformly to all kinds of infinite state systems. Due to space limitations, we will just briefly mention some of the promising current approaches.

Classically, Petri nets have been used to model concurrent processes using a single transition graph. More recently, verification methods for pushdown systems have been described [1, 17] which enable the direct modelling of unbounded calling stacks. An approach mainly geared at parameterized systems is regular model checking [4] where the infinite transition relation is described by finite automata, rendering many reachability properties decidable. Important progress in modelling dynamic data structures has been made in a three-valued framework [27]. A promising special form of predicate abstraction for hybrid systems has recently been proposed by Tiwari [29] who suggested the use of predicates describing analytical properties of functions such as $\frac{\mathrm{d}g}{\mathrm{d}x} > 0$.

4 Conclusion

In the twenty-five years since its invention, model checking has developed into a highly active research area of its own right which combines algorithms, logic, (discrete) mathematics and of course application knowledge. Although model checking is usually simpler to apply than theorem proving, it is still not always easy for engineers with the right application knowledge but without formal training in verification to use model checking to its full capability. As expressed

in Rushby's notion of "disappearing formal methods," we expect that for many settings model checking will finally become a push-button technology similar to compilers in which the trade-off between the preciseness and the computational cost of the correctness analysis can be controlled by a few simple parameters. Generally though, the principal undecidability of virtually all questions in software verification makes clear that there is no silver bullet for verification, and there will always be a need to design model checking methods specific to problem classes.

While verification of hardware and software systems will evidently remain a core concern of model checking, there are also exciting new avenues of research which often involve the combination of traditional model checking techniques with continuous mathematics, most notably the verification of stochastic, hybrid and biological systems.

Acknowledgments

The authors extend their gratitude to Stefan Katzenbeisser for his careful proofreading.

References

1. R. Alur, K. Etessami, and P. Madhusudan. A Temporal Logic of Nested Calls and Returns. In *Proc. Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, volume 2988 of LNCS, pages 467–481, 2004.
2. T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proc. Model Checking Software, 8th International SPIN Workshop*, volume 2057 of LNCS, pages 103–122, 2001.
3. A. Biere, A. Cimatti, E. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proc. 36th Conference on Design Automation (DAC)*, pages 317–320, 1999.
4. A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular Model Checking. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 403–418, 2000.
5. R.E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers* 35(8), pages 677–691, 1986.
6. J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. *In Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, 1990.
7. S. Chaki, E. Clarke, A. Groce, S. Jha, and H. Veith. Modular Verification of Software Components in C. In *Proc. 25th Int. Conference on Software Engineering (ICSE)*, pages 385–395, 2003. Extended version in IEEE Transactions on Software Engineering, 2004.
8. E. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Logics of Programs: Workshop*, volume 131 of *LNCS*, pages 52–71, 1981.

9. E. Clarke, T. Filkorn, S. Jha. Exploiting Symmetry In Temporal Logic Model Checking. *Proc. Computer Aided Verification (CAV)*, volume 697 of LNCS, pages 450–462, 1996.
10. E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 154–169, 2000. Extended version in *J. ACM* 50(5): 752–794, 2003.
11. E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994.
12. E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, Cambridge, MA, 1999.
13. E. Clarke and H. Schlingloff. Model checking. In J. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, pages 1367–1522. Elsevier, Amsterdam, 2000.
14. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. Symposium on Principles of Programming Languages (POPL)*, pages 238–252, 1977.
15. E. Emerson. Temporal and modal logic. In J. van Leeuven, editor, *Handbook of Theoretical Computer Science, Vol. B.*, pages 995–1072. Elsevier, Amsterdam, 1990.
16. E.A. Emerson and A.P. Sistla. Symmetry and model checking. *Proc. Computer Aided Verification (CAV)*, volume 697 of LNCS, pages 463–478, 1996.
17. J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient Algorithms for Model Checking Pushdown Systems. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV)*, volume 1855 of *LNCS*, pages 232–247, 2000.
18. P. Godefroid. Using partial orders to improve automatic verification methods. In *Proc. Computer Aided Verification (CAV)*, volume 531 of LNCS, pages 176–185, 1990.
19. S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *Proc. Computer Aided Verification (CAV)*, volume 1254 of LNCS, pages 72–83, 1997.
20. T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. In *Proc. ACM SIGPLAN-SIGACT Conference on Principles of Programming Languages*, pages 58–70, 2002.
21. G. J. Holzmann. *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, Reading, MA, 2003.
22. M. Huth and M. Ryan. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press, London, 1999.
23. R. P. Kurshan. *Computer-Aided Verification of Coordinating Processes*. Princeton University Press, Princeton, NJ, 1994.
24. K. McMillan. *Symbolic Model Checking: An Approach to the State Explosion Problem*. Kluwer Academic Publishers, Dordrecht, 1993.
25. A. Pnueli. The temporal logic of programs. In *Proc. 18th Symposium on Foundations of Computer Science (FOCS)*, pages 46–67, 1977.
26. J. Queille and J. Sifakis. Specification and verification of concurrent systems in CESAR. In *Proc. 5th Int. Symposium in Programming*, volume 137 of *LNCS*, pages 337–351, 1982.

27. M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. In *Proc. ACM Transactions on Programming Languages and Systems* 24, 3, pages 217–298, 2002.
28. C. Stirling. Bisimulation, Modal Logic and Model Checking Games. *Logic Journal of the IGPL*, 7(1), pages 103–124, 1999.
29. A. Tiwari and G. Khanna. Series of Abstractions for Hybrid Systems. In *Proc. 5th Int. Workshop on Hybrid Systems: Computation and Control (HSCC 2002)*, volume 2289 of LNCS, pages 465–478, 2002.
30. M. Y. Vardi and P. Wolper. Reasoning about infinite computations. In *Information and Computation*, 115(1): pages 1–37, 1994.

Switched Systems

Daniel Liberzon*

Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL 61821, U.S.A.
liberzon@uiuc.edu

1 Introduction

This chapter is concerned with dynamical systems described by a combination of ordinary differential equations and discrete switching events. Although systems theory has traditionally focused on either continuous or discrete behavior, many (if not most) of the dynamical systems encountered in practice involve both types of dynamics. Important classes of such systems are provided by *networked control systems*, in which information shared by continuous subsystems is updated in a discrete fashion, and *embedded systems*, in which computer software interacts with physical devices.

Dynamical systems that are described by an interaction between continuous and discrete dynamics are usually called *hybrid systems*. The field of hybrid systems has a strong interdisciplinary flavor, and different communities have developed different viewpoints. One approach, favored by researchers in computer science, is to concentrate on studying the discrete behavior of the system, while the continuous dynamics are assumed to take a relatively simple form. Basic issues in this context include well-posedness, simulation, and verification. Many researchers in systems and control theory, on the other hand, tend to regard hybrid systems as continuous systems with switching and place a greater emphasis on properties of the continuous state. The main issues then become stability analysis and control synthesis. The book [18] provides a good overview of both of these perspectives.

Switched systems are basically a result of considering hybrid systems from the latter point of view. To define more precisely what we mean by a switched system, consider a family $\{f_p : p \in \mathcal{P}\}$ of sufficiently regular functions from $\mathbb{R}^n$ to $\mathbb{R}^n$, parameterized by some index set $\mathcal{P}$. Let $\sigma : [0, \infty) \to \mathcal{P}$ be a piecewise constant function of time, called a *switching signal*. A *switched system* is then described by the differential equation

*Supported by NSF ECS-0134115 CAR, NSF ECS-0114725, and DARPA/AFOSR MURI F49620-02-1-0325 grants.

$$\dot{x} = f_\sigma(x). \tag{1}$$

One usually assumes here that the switching signal σ has a finite number of discontinuities—called *switching times* or simply *switches*—on every bounded time interval. The value of σ at a given time t might depend on t, or $x(t)$, or both, or may be generated by a more sophisticated mechanism involving memory (see [10] for details).

A complete hybrid model giving rise to (1) would involve a specification of dynamics governing the evolution of both x and σ, viewed as the continuous state and the discrete state of the system, respectively. Departing from this viewpoint, we neglect the details of the discrete behavior and instead consider all possible switching signals σ from a certain class. Thus switched systems can be viewed as higher-level abstractions of hybrid systems, although they are of interest in their own right. Typically, such an abstraction yields a system that is simpler to describe but possesses more solutions than the original system of interest. For a more detailed discussion of the relationship between switched and hybrid systems, see [6].

We note that, going one level of abstraction further, we arrive at the *differential inclusion*

$$\dot{x} \in \{f_p(x) : p \in \mathcal{P}\}.$$

Differential inclusions are well studied in the mathematical literature [1]. Loosely speaking, they have a higher degree of regularity than switched systems, achieved by allowing extra solutions. In particular, arbitrarily fast switching phenomena (known as *chattering* or *sliding modes*) are automatically covered by this framework; these issues are also discussed in [10, 18].

Implicit in (1) is the assumption that, at switching times, the state x remains continuous and only the velocity $\dot{x}$ undergoes an abrupt change. If we allow x to instantaneously jump to a different value, we obtain a switched system with *impulse effects*. Traditionally, *impulsive systems* are actually defined as systems with jumps in the state but not in the velocity [2]. It is clear that switched systems have many similarities to (and draw inspiration from) impulsive systems. Impulsive systems are directly relevant to networked control systems, where some variables are updated at the instants when new information arrives (see [11, 16] and the references therein).

In this chapter we concentrate on *analysis* issues for switched systems of the form (1). However, such analysis is heavily motivated by *control synthesis* questions arising in the field of *switching control*. Suppose that we are given a process, typically described by a continuous-time control system, and need to find a controller such that the closed-loop system displays a desired behavior. In some cases, this can be achieved by applying a continuous static or dynamic feedback control law. In other cases, a continuous feedback law that solves the problem may not exist. A possible alternative in such situations is to incorporate logic-based decisions into the control law and implement switching among a family of controllers. This yields a switched (or hybrid) closed-loop system. Classes of systems naturally treated by switching control

techniques include nonholonomic systems, systems with large-scale modeling uncertainty, and systems with sensor and/or actuator limitations. The last category is very broad and includes in particular networked and embedded control systems. The design of switching control strategies for the above system classes is addressed in [10].

As we mentioned earlier, our main concern will be with properties of the continuous state x of the switched system (1). What makes switched systems interesting and challenging is that such properties can be gained or lost as a result of switching. In other words, it is in general neither necessary nor sufficient that each of the individual subsystems

$$\dot{x} = f_p(x), \qquad p \in \mathcal{P} \tag{2}$$

have the property of interest.

In most of what follows, we take asymptotic stability (in the sense of Lyapunov) as a representative example of a desired property. Stability is the most fundamental and extensively studied issue in the literature on switched systems (and dynamical systems in general), and it is very suitable for illustrating the main difficulties. At the end of this chapter, we provide some remarks and references on other system properties.

2 The Stability Problem

We assume that the reader is familiar with basic concepts of Lyapunov's stability theory for continuous-time systems ([9] is a good reference). Stability definitions for switched systems are obtained by straightforward modification of the standard stability concepts for non-switched systems. Here they are always formulated with respect to the origin, which is assumed to be an equilibrium, and the initial time $t_0 = 0$. For example, given a fixed switching signal σ, we say that the switched system (1) is *stable* (in the sense of Lyapunov) if for every $\varepsilon > 0$ there exists a $\delta > 0$ such that

$$|x(0)| \leq \delta \quad \Rightarrow \quad |x(t)| \leq \varepsilon \qquad \forall\, t \geq 0.$$

We say that (1) is *globally asymptotically stable* if it is stable and for every pair of positive numbers ε and δ there exists a $T > 0$ such that

$$|x(0)| \leq \delta \quad \Rightarrow \quad |x(t)| \leq \varepsilon \qquad \forall\, t \geq T.$$

In particular, if there exist positive constants c and λ such that all solutions of (1) satisfy the inequality

$$|x(t)| \leq c|x(0)|e^{-\lambda t} \qquad \forall\, t \geq 0,$$

then (1) is called *globally exponentially stable*. Local versions of the last two properties can also be defined in the standard way.

Instead of just (asymptotic, exponential) stability for each particular switching signal, a somewhat stronger property is often desirable, namely, asymptotic or exponential stability that is *uniform* over the set of all switching signals (not to be confused with the more common usage which refers to uniformity with respect to the initial time for time-varying systems). This uniformity is defined by requiring that the values of the constants whose existence is stipulated in the above definitions (δ, T, c, and λ) be independent of the choice of the switching signal σ. The resulting notions of *global uniform asymptotic stability* and *global uniform exponential stability* will be abbreviated as GUAS and GUES, respectively. (In the case of constrained switching, it also makes sense to consider stability properties that are uniform over all switching signals from a certain class [6].)

To understand the issues arising in stability analysis of switched systems, consider the situation where $\mathcal{P} = \{1, 2\}$ and $x \in \mathbb{R}^2$, so that we are switching between two systems in the plane. First, suppose that the two individual subsystems are asymptotically stable, with trajectories as shown on the left in Fig. 1 (the solid curve and the dotted curve). For different choices of the switching signal, the switched system might be asymptotically stable or unstable (these two possibilities are shown in Fig. 1 on the right).

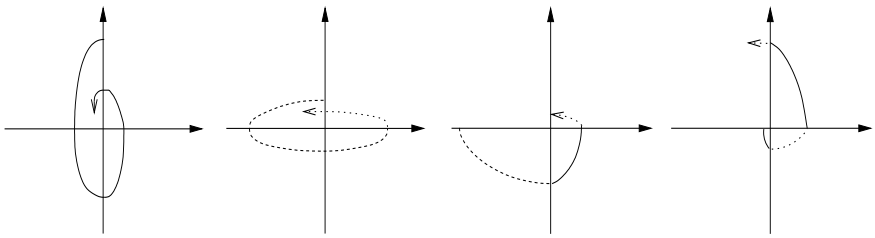


Fig. 1. Switching between stable systems

Similarly, Fig. 2 illustrates the case when both individual subsystems are unstable. Again, the switched system may be either asymptotically stable or unstable, depending on a particular switching signal. (We point out that interesting phenomena such as the ones demonstrated by these figures are only possible in dimensions 2 and higher.)

From these two examples, the following facts can be deduced:

- Unconstrained switching may destabilize a switched system even if all individual subsystems are stable.
- It may be possible to stabilize a switched system by means of suitably constrained switching even if some (or all) individual subsystems are unstable.

Motivated by these considerations, we will be studying the following two main problems:

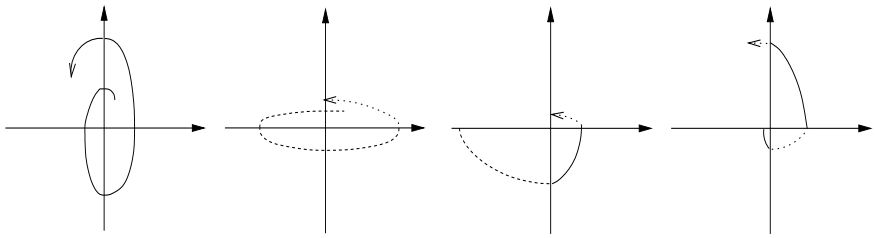


Fig. 2. Switching between unstable systems

(1) Stability under arbitrary switching: find conditions that guarantee (uniform) asymptotic stability of a switched system for arbitrary switching signals.
(2) Stability under constrained switching: if a switched system is not asymptotically stable for arbitrary switching, identify those switching signals for which it is asymptotically stable.

The first problem is relevant when the switching mechanism is unconstrained, unknown, or too complicated to be useful in the stability analysis. Stability under arbitrary switching is a very desirable property. When the subsystems being switched are obtained as feedback interconnections of a given process with different stabilizing controllers, it means that one does not need to worry about stability and can concentrate on other issues such as performance. While studying the first problem, one is led to investigate possible sources of instability, which in turn provides insight into the more practical second problem.

In the context of the second problem, it is natural to distinguish between two situations. If some or all of the individual subsystems are asymptotically stable, then it is of interest to characterize, as completely as possible, the class of switching signals that preserve asymptotic stability (such switching signals clearly exist; for example, just let $\sigma(t) \equiv p$, where p is the index of some asymptotically stable subsystem). On the other hand, if all individual subsystems are unstable, then the task at hand is to construct at least one stabilizing switching signal, which may actually be quite difficult or even impossible.

The latter problem is seen to be a synthesis problem and will not be treated in this chapter (see [10] for more information on this problem). We will thus assume throughout that all individual subsystems (2) are asymptotically stable. (This assumption is not always realistic for networked and embedded control systems, but a similar analysis applies when the total time spent in unstable modes is sufficiently small; see [10] for references.) The task is then to determine what additional requirements on the systems from (2) must be imposed to guarantee stability of the switched system (1). *Unless specified otherwise, original references and further details on the results discussed in the next two sections can be found in the book [10].*

3 Stability Under Arbitrary Switching

3.1 Common Lyapunov functions

Uniform stability properties of the switched system (1) are intimately related to the existence of a function that serves as a Lyapunov function for all individual subsystems (2). Given a positive definite continuously differentiable function $V : \mathbb{R}^n \to \mathbb{R}$, we will say that it is a *common Lyapunov function* for the family of systems (2) if there exists a positive definite continuous function $W : \mathbb{R}^n \to \mathbb{R}$ such that we have

$$\frac{\partial V}{\partial x} f_p(x) \leq -W(x) \qquad \forall\, x, \quad \forall\, p \in \mathcal{P}.$$

Theorem 1 *If all systems in the family* (2) *share a radially unbounded common Lyapunov function, then the switched system* (1) *is globally uniformly asymptotically stable (GUAS).*

This result is well known and can be derived in the same way as the standard Lyapunov stability theorem (cf. [9]). The main point is that the rate of decrease of V along solutions, characterized by W, is not affected by switching, hence asymptotic stability is uniform with respect to σ. In the special case when both V and W are quadratic (or, more generally, are bounded from above and below by monomials of the same degree in $|x|$), it is easy to show that the switched system is globally uniformly exponentially stable (GUES).

In the following, we will be concerned with identifying classes of switched systems that are GUAS. The most common approach to this problem consists of searching for a common Lyapunov function shared by the individual subsystems. A justification of this approach comes from the *converse Lyapunov theorem* for switched systems, which says that the GUAS property of a switched system implies the existence of a common Lyapunov function. For such a converse Lyapunov theorem to hold, the family of systems (2) needs to satisfy suitable technical conditions. We omit the details but mention that these conditions automatically hold when the index set $\mathcal{P}$ is finite.

A useful result, which can be derived from the converse Lyapunov theorem, says that if the switched system (1) is GUAS, then all "convex combinations" of the individual subsystems from the family (2) must be globally asymptotically stable. An informal interpretation of this result comes from the fact that one can mimic the behavior of the convex combination $\dot{x} = \alpha f_p(x) + (1-\alpha) f_q(x)$, $\alpha \in [0,1]$ by means of fast switching between the subsystems $\dot{x} = f_p(x)$ and $\dot{x} = f_q(x)$, spending the correct proportion of time (α versus $1-\alpha$) on each one. (For the same reason, the existence of a stable convex combination in the case of unstable subsystems leads naturally to the design of a stabilizing switching signal.) Stability of all convex combinations often serves as an easily checkable necessary condition for GUAS (it is not sufficient).

A particular case of interest is when all individual subsystems are linear, yielding a *switched linear system*

$$\dot{x} = A_\sigma x. \tag{3}$$

Then GUES is equivalent to the seemingly weaker property of local attractivity for every fixed switching signal. For switched linear systems, it is natural to consider *quadratic common Lyapunov functions*, i.e., functions of the form

$$V(x) = x^T P x, \tag{4}$$

where P is a positive definite symmetric matrix such that for some positive definite symmetric matrix Q we have

$$A_p^T P + P A_p \leq -Q \qquad \forall\, p \in \mathcal{P}. \tag{5}$$

One reason why quadratic common Lyapunov functions are attractive is that (5) is a system of *linear matrix inequalities* (LMIs), and there are efficient methods for solving finite systems of such inequalities numerically using tools from convex optimization. A general reference on LMIs is [3], and the survey paper [4] discusses them specifically in the context of switched linear systems. An alternative method for computing quadratic common Lyapunov functions is presented in [12]. It is in general not sufficient to work with quadratic common Lyapunov functions. However, GUES of a switched linear system can always be verified by a common Lyapunov function that is homogeneous of degree 2.

3.2 Commutation relations

The stability problem for switched systems can be studied from several different angles. We now explore a particular direction, namely, the role of commutation relations among the systems being switched.

Consider the switched linear system (3), and assume for the moment that $\mathcal{P} = \{1, 2\}$ and that the matrices A_1 and A_2 commute: $A_1 A_2 = A_2 A_1$. We can write the latter condition as $[A_1, A_2] = 0$, where the *commutator*, or *Lie bracket* $[\cdot, \cdot]$, is defined as

$$[A_1, A_2] := A_1 A_2 - A_2 A_1. \tag{6}$$

It is well known that in this case we have $e^{A_1} e^{A_2} = e^{A_2} e^{A_1}$. This means that the flows of the two individual subsystems $\dot{x} = A_1 x$ and $\dot{x} = A_2 x$ commute. Now consider an arbitrary switching signal σ, and denote by ρ_i and τ_i the lengths of the time intervals on which σ equals 1 and 2, respectively. The solution of the system produced by this switching signal is

$$\begin{aligned} x(t) &= \cdots e^{A_2\tau_2} e^{A_1\rho_2} e^{A_2\tau_1} e^{A_1\rho_1} x(0) = \cdots e^{A_2\tau_2} e^{A_2\tau_1} \cdots e^{A_1\rho_2} e^{A_1\rho_1} x(0) \\ &= e^{A_2(\tau_1+\tau_2+\cdots)} e^{A_1(\rho_1+\rho_2+\cdots)} x(0). \end{aligned}$$

Since at least one of the series $\rho_1 + \rho_2 + \cdots$ and $\tau_1 + \tau_2 + \cdots$ converges to ∞ as $t \to \infty$, the corresponding matrix exponential converges to zero in view of the stability of the matrices A_1 and A_2 (recall that asymptotic stability of individual subsystems is assumed). We have thus proved that $x(t) \to 0$ for an arbitrary switching signal. Generalization to the case when $\mathcal{P}$ has more than two elements is straightforward. Since for switched linear systems global attractivity for every σ implies GUES, we have the following result.

Theorem 2 *If $\{A_p : p \in \mathcal{P}\}$ is a finite set of commuting Hurwitz matrices, then the corresponding switched linear system* (3) *is GUES.*

There is also a more direct way to arrive at this result, which is based on constructing a common Lyapunov function for the family of linear systems

$$\dot{x} = A_p x, \qquad p \in \mathcal{P} \tag{7}$$

by means of an elegant iterative procedure.

To extend the above result to switched nonlinear systems, we first need the notion of a Lie bracket, or commutator, of two vector fields. This is the vector field defined as follows:

$$[f_1, f_2](x) := \frac{\partial f_2(x)}{\partial x} f_1(x) - \frac{\partial f_1(x)}{\partial x} f_2(x).$$

If the Lie bracket of two vector fields is identically zero, we will say that the two vector fields commute. The following result is a direct generalization of Theorem 2. Similarly to the linear case, it can be proved either by direct analysis of the flow or by an iterative construction of a common Lyapunov function.

Theorem 3 *If $\{f_p : p \in \mathcal{P}\}$ is a finite set of commuting vector fields and the origin is a globally asymptotically stable equilibrium for all systems in the family* (2)*, then the corresponding switched system* (1) *is GUAS.*

Another approach is to linearize the individual subsystems and apply the linear results together with Lyapunov's indirect method. If the linearization matrices are Hurwitz and commute, then a quadratic common Lyapunov function for the linearized systems, constructed as explained earlier, serves as a *local* common Lyapunov function for the original family of nonlinear systems (2).

Consider again the switched linear system (3). In view of the previous discussion, it is not surprising that when the matrices A_p, $p \in \mathcal{P}$ do not commute, stability of the switched system is still related to the commutation relations between them. A useful object which reveals the nature of these commutation relations is the *Lie algebra* $\mathfrak{g} := \{A_p : p \in \mathcal{P}\}_{LA}$ generated by the matrices A_p, $p \in \mathcal{P}$, with respect to the standard Lie bracket (6). This

is a linear vector space of dimension at most n^2, spanned by the given matrices and all their iterated Lie brackets. The structure of matrix Lie algebras guaranteeing GUES of the corresponding switched linear system has been completely characterized. The nonlinear case is much less explored, although some results going beyond Theorem 3 have recently been obtained in [14].

3.3 Systems with special structure

The stability problem for general switched systems is very difficult. Therefore, it is of interest to identify specific classes of systems for which some useful results can be obtained. For example, if $\{A_p : p \in \mathcal{P}\}$ is a compact set of Hurwitz matrices in triangular form, then the switched linear system (3) is GUES. This can be proved either directly, proceeding from the bottom component of x upwards, or by constructing a common Lyapunov function. It turns out that in this case, it is possible to find a quadratic common Lyapunov function of the form (4) with P a diagonal matrix. On the other hand, it can be shown by a counterexample that in the case of switching among globally asymptotically stable nonlinear systems, the triangular structure alone is not sufficient for GUAS.

As we explained in Section 1, switched systems often arise from the feedback connection of different controllers with the same process. Such feedback switched systems therefore assume particular interest in control theory. The fact that the process is fixed imposes some structure on the closed-loop systems, which sometimes facilitates the stability analysis.

For example, consider the system

$$\dot{x} = f(x, u), \qquad y = h(x)$$

and assume that it is (strictly) *passive*, in the sense that there exist a positive definite continuously differentiable function $V : \mathbb{R}^n \to \mathbb{R}$ (called a *storage function*) and a positive definite function $W : \mathbb{R}^n \to \mathbb{R}$ such that we have

$$\frac{\partial V}{\partial x} f(x, u) \leq -W(x) + u^T h(x).$$

It is easy to see that for every $K \geq 0$, the closed-loop system obtained by setting $u = -Ky$ is asymptotically stable, with Lyapunov function V. In other words, V is a common Lyapunov function for the family of closed-loop systems corresponding to all nonpositive definite feedback gain matrices. It follows that the switched system generated by this family is uniformly asymptotically stable (GUAS if V is radially unbounded). Clearly, the function V also serves as a common Lyapunov function for all nonlinear feedback systems obtained by setting $u = -\varphi(y)$, where φ satisfies $y^T \varphi(y) \geq 0$ for all y. In the single-input, single-output (SISO) case, this reduces to the sector condition

$$0 \leq y\varphi(y) \qquad \forall\, y.$$

For linear systems, the famous *Kalman–Yakubovich–Popov lemma* provides a frequency-domain characterization of passivity in terms of *positive realness* of the transfer matrix g. There is also a generalization, known as the *circle criterion*, which says that the above result holds if the function

$$\frac{1+k_2 g}{1+k_1 g}$$

is strictly positive real for some $k_2 > k_1 \geq 0$ and φ satisfies the more restrictive sector condition

$$k_1 y^2 \leq y\varphi(y) \leq k_2 y^2 \qquad \forall\, y.$$

We note that *Popov's criterion*, which leads to more powerful conditions for absolute stability of feedback systems, does not directly extend to switched systems in the same fashion, because it yields a Lyapunov function that explicitly depends on the nonlinearity (and thus does not provide a common Lyapunov function).

A different (but related) set of results is provided by the *small-gain theorem*. Consider the output feedback switched linear system

$$\dot{x} = (A + BK_\sigma C)x. \tag{8}$$

Assume that A is a Hurwitz matrix and that $\|K_p\| \leq 1$ for all $p \in \mathcal{P}$, where $\|\cdot\|$ denotes the matrix norm induced by the Euclidean norm on $\mathbb{R}^n$. Then the classical small-gain theorem implies that (8) is GUES if

$$\|C(sI - A)^{-1}B\|_\infty < 1,$$

where $\|\cdot\|_\infty$ denotes the standard H_∞ norm of a transfer matrix. This condition is satisfied if and only if there exists a solution $P > 0$ of the algebraic Riccati inequality

$$A^T P + PA + PBB^T P + C^T C < 0.$$

The function $V(x) = x^T Px$ then serves as a quadratic common Lyapunov function for the family of linear systems

$$\dot{x} = (A + BK_p C)x, \qquad p \in \mathcal{P}$$

and actually also for the family of nonlinear feedback systems

$$\dot{x} = Ax + B\varphi_p(Cx), \qquad p \in \mathcal{P},$$

provided that each φ_p satisfies

$$|\varphi_p(y)| \leq |y| \qquad \forall\, y.$$

Additional flexibility is gained if input-output properties of the process and the controllers are specified but one has some freedom in choosing state-space

realizations. In fact, given a strictly proper transfer matrix of the process and a finite family of transfer matrices of stabilizing controllers, there always exist realizations of the process and the controllers such that the corresponding closed-loop systems share a quadratic common Lyapunov function.

For switched linear systems in two dimensions, there are many results which rely on planar geometry (and do not have counterparts in higher dimensions). In particular, necessary and sufficient conditions for GUES and for the existence of a quadratic common Lyapunov function are available.

4 Stability Under Constrained Switching

4.1 Multiple Lyapunov functions

There is a useful tool for proving stability of switched systems which relies on *multiple Lyapunov functions*, usually one or more for each of the individual subsystems being switched. To fix ideas, consider the switched system (1) with $\mathcal{P} = \{1, 2\}$. Suppose that both systems $\dot{x} = f_1(x)$ and $\dot{x} = f_2(x)$ are (globally) asymptotically stable, and let V_1 and V_2 be their respective Lyapunov functions. We are interested in the situation where a common Lyapunov function for the two systems is not known or does not exist. In this case, one can try to investigate stability of the switched system using V_1 and V_2.

In the absence of a common Lyapunov function, stability properties of the switched system depend on the switching signal σ. Let t_i, $i = 1, 2, \ldots$ be the switching times. If it so happens that the values of V_1 and V_2 coincide at each switching time, i.e., $V_{\sigma(t_{i-1})}(t_i) = V_{\sigma(t_i)}(t_i)$ for all i, then V_σ is a continuous Lyapunov function for the switched system, and asymptotic stability follows. This situation is depicted in Fig. 3(left).

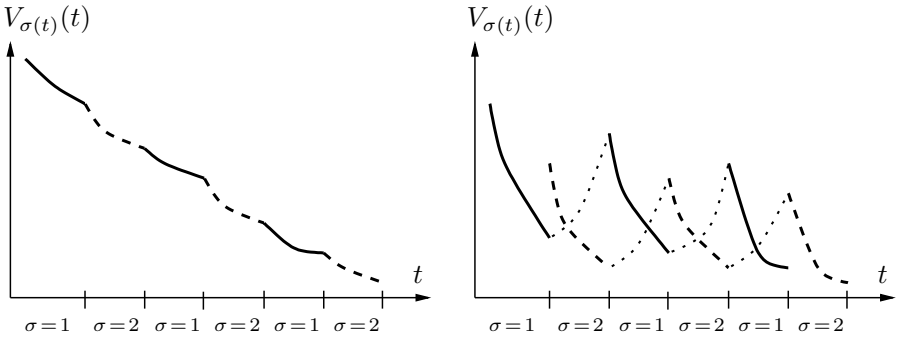


Fig. 3. Two Lyapunov functions (solid graphs correspond to V_1, dashed graphs correspond to V_2): (left) continuous V_σ, (right) discontinuous V_σ

In general, however, the function V_σ will be discontinuous. While each V_p decreases when the pth subsystem is active, it may increase when the pth

subsystem is inactive. This behavior is illustrated in Fig. 3(right). Let us look at the values of V_p at the beginning of each interval on which $\sigma = p$. The switched system is asymptotically stable if these values form a decreasing sequence for each p.

Theorem 4 *Let* (2) *be a finite family of globally asymptotically stable systems, and let* V_p, $p \in \mathcal{P}$ *be a family of corresponding radially unbounded Lyapunov functions. Suppose that there exists a family of positive definite continuous functions* W_p, $p \in \mathcal{P}$ *with the property that for every pair of switching times* (t_i, t_j), $i < j$ *such that* $\sigma(t_i) = \sigma(t_j) = p \in \mathcal{P}$ *and* $\sigma(t_k) \neq p$ *for* $t_i < t_k < t_j$, *we have*

$$V_p(x(t_j)) - V_p(x(t_i)) \leq -W_p(x(t_i)). \tag{9}$$

Then the switched system (1) *is globally asymptotically stable.*

Multiple Lyapunov function results such as Theorem 4 are useful when the class of admissible switching signals is constrained in a way that ensures the desired relationships between the values of Lyapunov functions at switching times. The situation described by Fig. 3(left) may arise in the case of *state-dependent switching*, i.e., if the switches are triggered when the state trajectory crosses some *switching surfaces*, or *guards*, in the state space. If the switching surfaces are constructed in such a way that the values of different Lyapunov functions on these surfaces coincide, then V_σ is indeed a continuous function of time. For switched linear systems, the corresponding conditions can also be cast as LMIs [4]. Stability analysis for state-dependent switching is also often facilitated by the fact that properties of each individual subsystem are of concern only in the regions where this system is active. For example, consider the first (stable) switching strategy in Fig. 1; it can be analyzed by a single Lyapunov function, even though a common Lyapunov function does not exist (as is clear from the existence of the second, unstable switching strategy).

The more general situation shown in Fig. 3(right) is capable of covering the case of *time-dependent switching*, i.e., switching signals having certain time patterns not affected by the state. It is well known that a switched system is stable if the switching is sufficiently slow, so as to allow the transient effects to dissipate after each switch. In the next subsection we discuss how such slow switching conditions can be formulated and justified using multiple Lyapunov functions. These techniques are relevant for networked control systems, because they help characterize the relationship between the information rate and stability. They are also important for switching control design, where slow switching conditions can be explicitly incorporated into—or derived from—the switching logic.

4.2 Stability under slow switching

The simplest way to specify slow switching is to introduce a number $\tau_d > 0$ and restrict the class of admissible switching signals by demanding that the

switching times t_1, t_2, ... satisfy the inequality $t_{i+1} - t_i \geq \tau_d$ for all i. This number τ_d is usually called the *dwell time* (because σ "dwells" on each of its values for at least τ_d units of time).

When all linear systems in the family (7) are asymptotically stable, the switched linear system (3) is asymptotically stable if the dwell time τ_d is sufficiently large. The required lower bound on τ_d can be explicitly calculated from the exponential decay bounds on the transition matrices of the individual subsystems (see, e.g, [15, Lemma 2]). Under suitable assumptions, a sufficiently large dwell time also guarantees asymptotic stability of the switched system in the nonlinear case. Probably the best way to prove most general results of this kind is by using multiple Lyapunov functions.

To see how this works, consider the example where $\mathcal{P} = \{1, 2\}$ and both systems in the family (2) are globally exponentially stable. Then there exist Lyapunov functions V_1 and V_2 which for some positive constants a, b, and c satisfy

$$a|x|^2 \leq V_p(x) \leq b|x|^2 \tag{10}$$

and

$$\frac{\partial V_p}{\partial x} f_p(x) \leq -c|x|^2 \tag{11}$$

for $p = 1, 2$. Combining (10) and (11), we obtain

$$\frac{\partial V_p}{\partial x} f_p(x) \leq -2\lambda V_p(x), \qquad p = 1, 2,$$

where $\lambda := c/2b$. This implies that

$$V_p(x(t_0 + \tau_d)) \leq e^{-2\lambda\tau_d} V_p(x(t_0))$$

whenever $\sigma(t) = p$ for $t \in [t_0, t_0 + \tau_d)$. Suppose that σ takes the value 1 on $[t_0, t_1)$ and 2 on $[t_1, t_2)$, where $t_{i+1} - t_i \geq \tau_d$, $i = 0, 1$. From the preceding inequalities we have

$$V_1(t_2) \leq \frac{b_1}{a_2} V_2(t_2) \leq \frac{b_1}{a_2} e^{-2\lambda_2\tau_d} V_2(t_1) \leq \frac{b_1 b_2}{a_1 a_2} e^{-2(\lambda_1+\lambda_2)\tau_d} V_1(t_0).$$

It is now straightforward to compute an explicit lower bound on τ_d which guarantees that the hypotheses of Theorem 4 are satisfied, implying that the switched system (1) is globally asymptotically stable:

$$\tau_d > \frac{1}{2(\lambda_1 + \lambda_2)} \log \frac{b_1 b_2}{a_1 a_2}.$$

In the context of controlled switching, specifying a dwell time may be too restrictive. If, after a switch occurs, there can be no more switches for the next τ_d units of time, then it is impossible to react to possible system failures or unacceptable performance during that time interval. Thus it is of interest to relax the concept of dwell time, allowing the possibility of switching fast

when necessary and then compensating for it by switching sufficiently slowly later.

The concept of average dwell time from [8] serves this purpose. Let us denote the number of discontinuities of a switching signal σ on an interval (t, T) by $N_\sigma(T, t)$. We say that σ has *average dwell time* τ_a if there exist two positive numbers N_0 and τ_a such that

$$N_\sigma(T, t) \leq N_0 + \frac{T - t}{\tau_a} \qquad \forall\, T \geq t \geq 0. \tag{12}$$

For example, if $N_0 = 1$, then (12) implies that σ cannot switch twice on any interval of length smaller than τ_a. Switching signals with this property are exactly the switching signals with dwell time τ_a. In general, if we discard the first N_0 switches, then the average time between consecutive switches is at least τ_a.

It turns out that the property discussed earlier—namely, that asymptotic stability is preserved under switching with a sufficiently large dwell time—can be extended to switching signals with average dwell time. However, for switched nonlinear systems this involves additional constraints on the multiple Lyapunov functions. Under suitable conditions, a useful class of hysteresis-based switching logics is known to guarantee the existence of an average dwell time. See [8, 10] for details.

4.3 Results of LaSalle type

In the foregoing, we have been working with Lyapunov functions which are strictly decreasing along solutions. Another possibility is to work with *weak Lyapunov functions*, i.e., functions that are merely nonincreasing along solutions. The well-known *LaSalle's invariance principle* utilizes such functions [9]. It implies, in particular, that the system $\dot{x} = f(x)$ is globally asymptotically stable if there exists a positive definite, radially unbounded, continuously differentiable function $V : \mathbb{R}^n \to \mathbb{R}$ whose derivative along solutions satisfies $\frac{\partial V}{\partial x} f(x) \leq -W(x) \leq 0$ for all x, and if moreover the largest positively invariant set contained in the set $\{x : W(x) = 0\}$ is equal to $\{0\}$.

The second condition can be regarded as *observability* with respect to the auxiliary output $y := W(x)$. One generalization of this result to switched linear systems goes as follows.

Theorem 5 *Consider the family of linear systems* (7) *with* $\mathcal{P}$ *a finite set and a switching signal* σ *with switching times* t_i, $i = 1, 2, \ldots$. *Assume that the following conditions hold:*

1. *For each* $p \in \mathcal{P}$ *there exists a positive definite symmetric matrix* P_p *satisfying*
$$A_p^T P_p + P_p A_p \leq -C_p^T C_p$$
for some matrix C_p *such that* (C_p, A_p) *is an observable pair.*

2. *There exists a* $\tau > 0$ *such that for every* $T \geq 0$ *we can find a positive integer* i *for which* $t_{i+1} - \tau \geq t_i \geq T$.
3. *For each* $p \in \mathcal{P}$ *and every pair of switching times* $t_i < t_j$ *such that* $\sigma(t_i) = \sigma(t_j) = p$, *we have*

$$x^T(t_j)P_p x(t_j) \leq x^T(t_{i+1})P_p x(t_{i+1}).$$

Then the switched linear system (3) *is globally asymptotically stable.*

For further results and discussion on the linear case, see [6]. The recent paper [7] presents an extension of the above result to switched nonlinear systems and non-quadratic weak Lyapunov functions, which relies on a suitable nonlinear observability notion and which yields as a corollary a version of Popov's criterion for switched feedback systems. The usefulness of these results stems in part from the fact that it is sometimes easier to find weak Lyapunov functions nonincreasing along solutions and satisfying conditions such as the last one in Theorem 5 (or even a common weak Lyapunov function for a given family of systems) than to find strictly decreasing Lyapunov functions satisfying the condition (9) of Theorem 4 (or, in particular, a common Lyapunov function). Another version of LaSalle's theorem for hybrid systems, which uses a single weak Lyapunov function, appeared in [20].

5 Other Concepts

In this chapter, we took stability as the most representative and well-studied property to highlight the main issues. There are many other concepts of interest besides stability, especially for switched systems with inputs and/or outputs. The classical Kalman controllability and observability conditions can be extended to switched linear systems; see the references in [10] as well as the recent survey [17]. Input/output (or input/state) properties characterizing robustness of switched systems to disturbances are especially relevant for networked and embedded control systems. References on this subject include [5, 13, 19]. These and other properties of switched systems are currently under active investigation, and the literature on the subject is rapidly growing.

References

1. J.-P. Aubin and A. Cellina. *Differential Inclusions: Set-Valued Maps and Viability Theory*. Springer, Berlin, 1984.
2. D. D. Bainov and P. S. Simeonov. *Systems with Impulse Effect: Stability, Theory and Applications*. Ellis Horwood, England, 1989.
3. S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*, volume 15 of *SIAM Studies in Applied Mathematics*. SIAM, Philadelphia, 1994.

4. R. A. DeCarlo, M. S. Branicky, S. Pettersson, and B. Lennartson. Perspectives and results on the stability and stabilizability of hybrid systems. *Proc. IEEE*, 88:1069–1082, 2000.
5. J. P. Hespanha. Root-mean-square gains of switched linear systems. *IEEE Trans. Automat. Control*, 48:2040–2045, 2003.
6. J. P. Hespanha. Uniform stability of switched linear systems: Extensions of LaSalle's invariance principle. *IEEE Trans. Automat. Control*, 49:470–482, 2004.
7. J. P. Hespanha, D. Liberzon, D. Angeli, and E. D. Sontag. Nonlinear observability notions and stability of switched systems. *IEEE Trans. Automat. Control*, 2005. To appear.
8. J. P. Hespanha and A. S. Morse. Stability of switched systems with average dwell-time. In *Proc. 38th IEEE Conf. on Decision and Control*, pages 2655–2660, 1999.
9. H. K. Khalil. *Nonlinear Systems*. 3rd edition, Prentice Hall, Upper Saddle River, NJ, 2002.
10. D. Liberzon. *Switching in Systems and Control*. Birkhäuser, Boston, 2003.
11. D. Liberzon and J. P. Hespanha. Stabilization of nonlinear systems with limited information feedback. *IEEE Trans. Automat. Control*, 2005. To appear.
12. D. Liberzon and R. Tempo. Switched systems, common Lyapunov functions, and gradient algorithms. *IEEE Trans. Automat. Control*, 49:990–994, 2004.
13. J. L. Mancilla-Aguilar and R. A. Garcia. On converse Lyapunov theorems for ISS and iISS switched nonlinear systems. *Systems Control Lett.*, 42:47–53, 2001.
14. M. Margaliot and D. Liberzon. A Lie-algebraic condition for stability of switched nonlinear systems. In *Proc. 43rd IEEE Conf. on Decision and Control*, 2004.
15. A. S. Morse. Supervisory control of families of linear set-point controllers, Part 1: Exact matching. *IEEE Trans. Automat. Control*, 41:1413–1431, 1996.
16. D. Nešić and A. R. Teel. L_p stability of networked control systems. In *Proc. 42nd IEEE Conf. on Decision and Control*, pages 1188–1193, 2003. Full version to appear in *IEEE Trans. Automat. Control*.
17. Z. Sun and S. S. Ge. Analysis and synthesis of switched linear control systems. *Automatica*, 2005. To appear.
18. A. van der Schaft and H. Schumacher. *An Introduction to Hybrid Dynamical Systems*. Springer, London, 2000.
19. W. Xie, C. Wen, and Z. Li. Input-to-state stabilization of switched nonlinear systems. *IEEE Trans. Automat. Control*, 46:1111–1116, 2001.
20. J. Zhang, K. H. Johansson, J. Lygeros, and S. S. Sastry. Dynamical systems revisited: Hybrid systems with Zeno executions. In N. Lynch and B. H. Krogh, editors, *Proc. 3rd Int. Workshop on Hybrid Systems: Computation and Control*, volume 1790 of *Lecture Notes in Computer Science*, pages 451–464. Springer, Berlin, 2000.

Feedback Control with Communication Constraints *

Dimitrios Hristu-Varsakelis

Department of Applied Informatics,
University of Macedonia, Thessaloniki, 54006, Greece
dcv@uom.gr

1 Introduction

One of systems theory's most useful and fundamental ideas is that of interconnecting simple systems in order to build complex ones. This is usually accomplished through the use of two important tools. One is a set of theoretical results that help predict the behavior and performance of the composed system given the properties of its components and the manner in which they are connected. The other is the ability to regard the interconnection as ideal in the sense that it neither corrupts nor delays data or—in situations where that is not the case—to "separate" its design from that of the other components (e.g., controllers).

The development in recent years of embedded and network technologies has given rise to the area of *Networked Control Systems* (NCSs), where sensors, actuators and computing elements are connected by means of a network or other shared medium. At the same time, the attempt to expand the scope of systems theory into this new domain has made the assumptions stated above increasingly difficult to justify. The goal of this chapter is to expose some of the complications that arise when a control system includes a network (taken to mean a shared communication medium in the most generic sense) and to introduce a small collection of basic results on the control of systems that operate under communication constraints.

The very technologies that enable one to construct NCSs impose limitations in communication that make the interconnection of components nontrivial from the point of view of control. Some of the issues that arise include

- Delays in transmitting information between components (e.g., from a sensor to a controller). These delays could be fixed or time varying (e.g., randomly distributed).

*This work was supported by the National Science Foundation under Grant No. EIA0088081 and by ARO ODDR&E MURI01 Grant No. DAAD19-01-1-0465 (Center for Communicating Networked Control Systems, through Boston University).

- The possibility of data failing to reach its destination (this is not only a function of the communication medium but also of the protocol being used; TCP/IP is a well-known example).
- Bottlenecks; they could occur because the shared network can only accommodate a limited number of simultaneous communications between components or has limited throughput. Bottlenecks could also occur because of computational constraints, e.g., the CPU on which the controller is implemented can only perform a limited amount of computation per unit time.

These constraints can be captured mathematically through a variety of techniques, some of which will be reviewed in the sequel. However, the existence of the constraints has the effect of complicating what are otherwise well-understood control problems (e.g., stabilization, estimation, linear quadratic regulator (LQR) tracking and others). The basic mechanism by which this occurs will become clear in the development; for now it could be summarized by stating that when a control system is subject to communication constraints, the policies that govern how the communication medium is used can have a direct effect on the design of the control policy and vice versa. In the same setting, optimal control must now be regarded jointly with optimal communication, and the goal is to simultaneously optimize the controller and communication policies governing the operation of an NCS, whenever possible. In cases where that may be difficult, one may attempt to make the problem easier to solve by assuming, for example, that the communication policy is fixed while designing a controller or vice versa.

Possible responses to these challenges include amending existing theoretical tools to apply to the new domain and developing new ones from first principles. Details such as the communication protocol and the operating system of the computer on which control is implemented can also influence the design of both control and communication policies. Here we will focus on how communication constraints affect the control and omit the implementation details, which are nevertheless discussed in other chapters of this book.

In the next sections we will give an overview of some of the available theoretical tools for addressing analysis and design problems involving NCSs, and for elucidating the interaction between control and communication decisions in systems with limited communication. We will focus mainly on stabilization and estimation. We begin by outlining a basic model for NCSs before going on to discuss (in Section 3) some feedback control problems for NCS. The basic viewpoint is that of sensor and actuator elements competing for the "attention" (in the form of time on the shared network) of a remote controller. The effects of transmission delays and dropped packets are outlined in Sections 3.3 and 3.4. Section 4 reviews basic results on feedback control and estimation of NCSs, this time emphasizing bit rates (instead of time) as the measure of "attention."

2 A Basic Model of Networked Control Systems

Fig. 1 depicts a generic NCS; the system consists of a plant, controller and network across which all sensor and actuator data must be sent. We use $u(\cdot) \in$

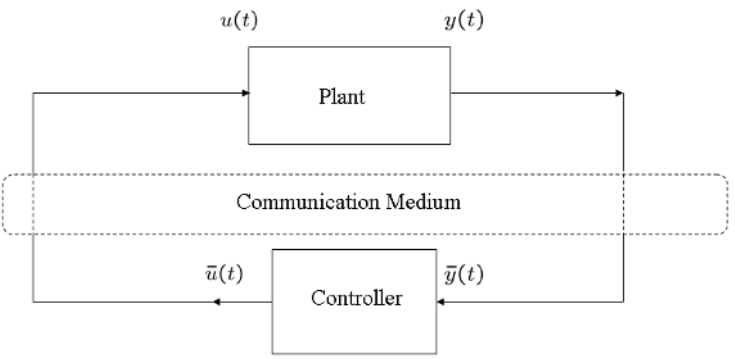


Fig. 1. A basic NCS, showing the underlying plant, its controller and the communication network that connects them

$\mathbb{R}^m$ and $y(\cdot) \in \mathbb{R}^p$ to denote the input and output of the plant, respectively. The quantities $\bar{y}(\cdot)$ and $\bar{u}(\cdot)$ denote the input and output of the controller, respectively. In general, these will differ from y and u because of the presence of the network. For example, u may be a delayed version of $\bar{u}$, if the network imposes only a delay. If the network cannot simultaneously carry signals for all m actuators, then some of the elements of u may be outdated compared to $\bar{u}$. Finally, if signals are quantized before being transmitted it may be that different elements of the vector u are quantized versions of the elements of $\bar{u}$ but with different accuracies. From a control design viewpoint, these considerations raise important questions like: "which sensor (actuator) should receive the most attention (in terms of time, frequency of communication or bit rate) by the controller?"

3 Modeling Medium Access Constraints

For now, we will ignore any transmission delays and quantization effects associated with controller/plant communication and focus instead on the bottlenecks created by the inability of the network to accommodate all sensors and actuators simultaneously. If transmission of a single sensor measurement takes t_s seconds, one may choose to "packetize" data from all sensors and transmit them every $p \cdot t_s$ seconds (what we refer to as *single-packet transmission*) or to sample one sensor at a time with frequency $1/t_s$ (*multiple packet transmission*). In the latter case, some sensors and actuators have access to the

controller while others wait. This situation is illustrated in Fig. 2, where two sets of switches control access to the communication medium. Let the plant

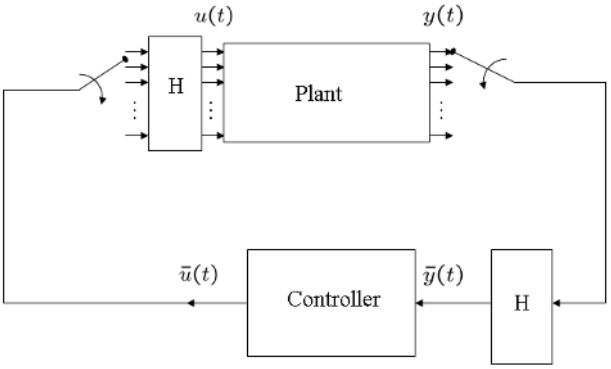


Fig. 2. Switch model

be linear time invariant (LTI), evolving in discrete time (the last assumption is not essential but it will simplify the discussion to follow):

$$x(k+1) = Ax(k) + Bu(k); \quad x \in \mathbb{R}^n, u \in \mathbb{R}^m \tag{1}$$

$$y(k) = Cx(k); \quad y \in \mathbb{R}^p. \tag{2}$$

Suppose that the communication medium connecting the sensors to the controller has n_σ $(1 \leq n_\sigma < p)$ *output channels*. At any one time, only n_σ of p sensors can access these channels to send their output to the controller, while others have to wait. Likewise, actuators share n_ρ $(1 \leq n_\rho < m)$ *input channels* to communicate with the controller, and at most n_ρ of them can do so simultaneously.

Of course, when a sensor (actuator) temporarily stops communicating with the controller (plant), the latter must decide how to handle the interruption. This takes place in the blocks denoted by H in Fig. 2. One option is for H to implement a zero-order hold (ZOH) so that the receiver uses the most recently transmitted value until communication is re-established. This has some appealing aspects but may increase the complexity of the control problem as we shall see. Another possibility is for the receiver to "ignore" the sensors or actuators that have gone off-line, in a way which will be made precise below.

The communication status of each sensor at time k can be encoded in the binary-valued function $\sigma_i(k), i = 1, ..., p$ with $\sigma_i(k) : \mathbb{Z} \mapsto \{0, 1\}$, where 1 means "accessing" and 0 means "not accessing". This leads to the following intuitive definition [8, 11].

Definition 1. *An m-to-n communication sequence is a map* $\sigma(k) : \mathbb{Z} \mapsto \{0, 1\}^m$, *satisfying* $\|\sigma(k)\|^2 = n$, $\forall k$.

The medium access status of the plant's sensors and actuators can then be represented by a pair of p-to-n_σ and m-to-n_ρ communication sequences, labeled σ and ρ, respectively. We will use σ (referred to as the *output communication sequence*) and ρ (the *input communication sequence*) to denote the sequences that govern the transmission of sensor and actuator data, respectively.

One is now faced with the problem of designing a pair of communication sequences and a controller that together achieve a desired control objective (e.g., stability). We will refer to the simultaneous selection of controller and communication sequence(s) for an NCS as the *joint problem*. We distinguish between two kinds of communication policies: *static* (or *fixed*), where a communication sequence is determined off-line, and *dynamic* (or *feedback based*), where communication decisions depend on the plant's outputs and on the access status of sensors and actuators.

Remark 1 (Selection of effective communication sequences). In general, the joint problem is difficult to solve when it comes to instances of typical NCS design problems, including stabilization and LQR tracking. When the joint problem is intractable, there are several alternatives:

- A typical approach is to postulate a communication sequence and then obtain a controller that satisfies the desired criteria. Such is the approach in Section 3.1 for example.
- Under some formulations, it is possible to narrow down the set of acceptable communication sequences and choose from that set. Sections 3.1 and 3.2 offer examples of this approach.
- Another alternative is to use heuristics or approximation methods in order to construct sequences that perform "sufficiently well" [18, 23].
- Finally, one could forgo the problem of choosing specific communication sequences and instead propose a policy for determining the communication on-line (as a function of time and sensor data, for example). We will discuss this further in Section 3.2.

3.1 Stability with a static communication sequence

We first consider the following problem

Problem 1. For an NCS whose plant is governed by (2) and whose controller can communicate with n_ρ and n_σ actuators and sensors respectively at any one time, find a pair of communication sequences σ, ρ, and a feedback controller $\bar{u}(k) = \Gamma(k)\bar{y}(k)$ so that the closed loop NCS is stable.

The solution to this problem is simplest if the controller and plant choose to "ignore" sensors and actuators which are not actively communicating, by assuming that the value of the corresponding output/input is simply zero. In that case, $\bar{y}(k)$, the output as seen by the controller is related to the actual output y by

$$\bar{y}(k) = \text{diag}(\sigma(k)) \cdot y(k), \tag{3}$$

where for $v \in \mathbb{R}^n$, $\text{diag}(v) \in \mathbb{R}^{n \times n}$ is the diagonal matrix formed using the elements of v. A similar relationship holds for $\bar{u}$, u and the input communication sequence ρ, so that from the point of view of the controller, the plant to be controlled is now time-varying:

$$x(k+1) = Ax(k) + B\text{diag}(\rho(k))\bar{u}(k) \tag{4}$$

$$\bar{y}(k) = \text{diag}(\sigma(k))Cx(k). \tag{5}$$

The stabilization problem can now be solved as follows:

- Restrict the solution to periodic communication sequences, so that the closed-loop dynamics (4) are periodic.
- Choose a periodic input (output) sequence that preserves the reachability (observability) of the plant. This is always possible if the original plant is controllable (observable) and A is invertible (as would be the case if (2) were obtained by discretizing a continuous time plant).
- Construct a periodic stabilizing feedback controller [24].

Theorem 1 ([30]). *Suppose A is invertible and the pair (A, B) of the plant (1) is reachable. For any integer $1 \leq n_\rho < m$, there exist integers $l, N > 0$ and an N-periodic p-to-n_ρ communication sequence ρ such that the extended plant (4) is l-step reachable, i.e., reachable on $[i, i+l]$ for any i.*

A communication sequence that preserves reachability can be easily constructed by examining the columns of

$$R = [A^{N-1}B \cdot \text{diag}(\rho(0)),\ A^{N-2}B \cdot \text{diag}(\rho(1)),\ \cdots\ B \cdot \text{diag}(\rho(N-1))]. \tag{6}$$

An algorithm is given in [30]; similar statements hold for observability. If state feedback is available ($C = I$, so that we can write $\bar{x} = \bar{y}$ and $y = x$) then we have the following.

Theorem 2 ([30]). *Suppose that the extended plant (4) is l-step reachable and that A is invertible. Given constants $\alpha > 1$, $\eta > 1$ the feedback controller $u(k) = \Gamma(k)\bar{x}(k)$, with*

$$\Gamma(k) = -\bar{B}^T(k)(A^{-1})^T \mathcal{W}_{\eta\alpha}^{-1}(k, k+l), \tag{7}$$

is such that the closed loop NCS is uniformly exponentially stable [24] with rate α, where $\bar{B}(k) = B \cdot \text{diag}(\rho(k))$ and

$$\mathcal{W}_\alpha(k_0, k_f) = \sum_{j=k_0}^{k_f-1} \alpha^{4(k_0-j)} A^{k_0-j-1} \bar{B}(j)\bar{B}^T(j)(A^{k_0-j-1})^T.$$

For the case of output feedback ($C \neq I$), the controller must be preceded by a state observer designed to reconstruct the plant's state from the intermittently arriving sensor data. The observer's state $\hat{x}(k)$ is then used in lieu of $\bar{x}(k)$ in the feedback controller. The (periodic) observer gains are selected using a procedure similar to that for selecting $\Gamma(k)$ (see [30] for details).

The effects of a ZOH

The stabilization problem becomes significantly more complicated if a ZOH is used when a sensor (actuator) relinquishes the network. In that case, the feedback controller has access only to $\bar{y}(k)$ (see Fig. 1), a vector composed of the most up-to-date sensor data available at the kth step. As we have mentioned, $\bar{y}(k) \neq y(k)$ because not all elements of $y(k)$ can be communicated to the controller at time k. A similar situation holds for u and the signal that actually arrives at the plant, $\bar{u}$. The communication sequences used at the input and output stages of the plant determine which components of u and y are updated at each time step. This leads to closed-loop dynamics of the form [9]

$$x(k+1) = Ax(k) + \sum_{i=0}^{2N-2} F_{ki}x(k-i), \tag{8}$$

where, assuming a constant feedback gain Γ ($\bar{u}(k) = \Gamma\bar{y}(k)$),

$$F_{ki} \triangleq B \sum_{j=\min(i,N-1)}^{\lfloor \frac{i}{N} \rfloor (i-N-1)} D_W(k,j)\Gamma D_R(k-j,i-j)C \tag{9}$$

and

$$D_R(k,i) \triangleq \begin{cases} \operatorname{diag}(\rho(k)) & i = 0 \\ \operatorname{diag}(\rho(k-i)) \prod_{j=0}^{i-1} M_R(k,j) & i > 0 \end{cases} \tag{10}$$

$$D_W(k,i) \triangleq \begin{cases} \operatorname{diag}(\sigma(k)) & i = 0 \\ \operatorname{diag}(\sigma(k-i)) \prod_{j=0}^{i-1} M_W(k,j) & i > 0 \end{cases} \tag{11}$$

with $M_R(k,j) \triangleq I - \operatorname{diag}(\rho(k-j))$, $M_W(k,j) \triangleq I - \operatorname{diag}(\sigma(k-j))$.

If the communication is periodic in k then so are the parameters F_{ki}, and (8) can be written in first-order form as [9, 10]

$$\chi(k+1) = \mathcal{F}_k\chi(k), \tag{12}$$

where $\chi = [x_{(k-2N+1)}^T \ \cdots \ x_{(k)}^T \ x_{(k+1)}^T]^T \in \mathbb{R}^{(2N-1)n}$. Equation (12) is linear time-varying, and describes the state evolution of the computer-controlled system under output feedback and N-periodic communication. The new state vector χ now includes past state values up to two communication periods. The periodic form (12) can be rewritten as a time-invariant system of higher dimension (equal to $N(2N-1)n$) to obtain what is known as the "extensive form" [9, 10] of the original system:

$$\mathcal{X}_e(k+1) = \mathcal{A}\mathcal{X}_e(k); \quad \mathcal{X}_e(k) \in \mathbb{R}^{(2N^2-N)n}, \tag{13}$$

where $\mathcal{A}$ is affine in the entries of the feedback gain Γ. For fixed Γ, the problem of selecting gains to guarantee stability is non-deterministic polynomial-time

hard (NP-hard) [2,10], even for a fixed communication sequence. The work in [10] describes a numerical approach to the problem, using simulated annealing to choose Γ so that the eigenvalues of $\mathcal{A}$ are enclosed in a circle with the smallest possible radius.

If we allow for time-varying feedback gains and assume state feedback, then stabilizing gains can be designed for the periodic form of the NCS (12) using results from linear periodic systems [15, 24]. On the other hand, the output feedback case, as well as the problem of simultaneously designing the communication sequences and controller, is not easy to approach. Some interesting special cases include [14]; that work discusses the stabilization of NCSs with time-varying decentralized controllers and gives criteria for stabilizability and rules for sequence design, although the latter problem becomes complex as the length of the sequence and number of possible interconnections grows.

3.2 Feedback-based communication

Feedback-based communication offers a sometimes attractive alternative to the problem of selecting communication sequences for NCSs. The idea is to let the position of the switches in Fig. 2 be determined by the state (or output) of the NCS by defining a suitable mapping

$$\sigma(x,t) \in \{0,1\}^m, \quad \sigma : \mathbb{R}^n \times \mathbb{R}_+ \to \{0,1\}^m \tag{14}$$

for the output sequence σ, and another for the input sequence ρ. In contrast to Section 3.1 where the controller and plant poll each other's outputs, here communication is interrupt driven. Such a choice has an obvious potential advantage: if the policy σ is chosen carefully, the controller may be able to respond immediately to changes in a sensor's output if they are deemed important. Under static communication, that sensor would have to wait for its turn, which could come much later, depending on the particular communication sequence chosen. On the other hand, static communication can guarantee that every sensor and actuator will be polled. This offers a robustness advantage, because it makes a "dead" sensor easy to detect, for example. Next, we give two examples of dynamic communication policies.

A block-diagonal NCS

Consider a collection of continuous-time linear time-invariant (LTI) systems

$$\begin{aligned} &\dot{x}_i(t) = A_i x_i(t) + B_i u_i(t); \quad i = 1, \ldots, N \\ &x_i(t) \in \mathbb{R}^n, u_i(t) \in \mathbb{R}^m \end{aligned} \tag{15}$$

whose open-loop dynamics are unstable ($\mathrm{Re}\{\lambda(A_i)\} > 0, \quad i = 1, \ldots, N$). Each system communicates with a remotely located controller over an idealized

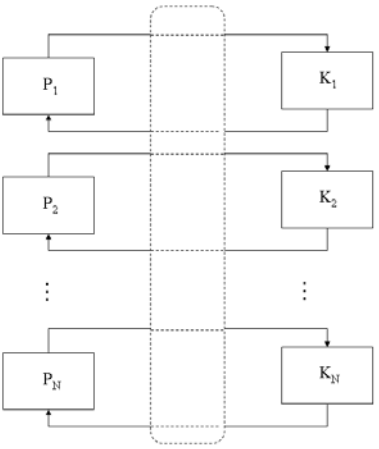


Fig. 3. A collection of NCSs $G_i(s) = I(sI - A_i)^{-1}B_i$ driven by static feedback controllers K_i via a network. Only k of N switches s_i can be closed at any one time [12].

shared network, according to the static state feedback law[1] $u_i(t) = K_i x_i(t)$ (see Fig. 3).

The gains K_i are *designed a priori* so that $\mathrm{Re}\{\lambda(A_i + B_i K_i)\} < 0$, $i = 1, \ldots, N$; i.e., each system is stabilized in the absence of communication constraints. Controller-plant communication is limited in the sense that a maximum of $C < N$ plants may close their feedback loops at any one time. We note that although there are no coupling terms in (15), the dynamics of the systems *are* coupled because of the presence of the communication constraint; if a system monopolizes the network others may not be stabilizable.

Problem 2. Find a *feedback-based* policy for establishing and terminating communication between each system and its controller in a way that stabilizes all systems in the collection.

To proceed, write the dynamics of each system in the collection as

$$\dot{x}_i(t) = A_{s_i(t)} x_i(t); \quad i = 1, \ldots, N, \tag{16}$$

where $A_{s_i(t)} \in \{A_i^o, A_i^c\}$, $A_i^o \triangleq A_i$ and $A_i^c \triangleq A_i + B_i K_i$ denote the open- and closed-loop dynamics of the collection, and $s_i(t) \in \{0, 1\}$ are piecewise constant functions that indicate when the ith loop is closed ($s_i(t) = 1$).

The following result [11] gives a sufficient condition for the existence of communication sequences that simultaneously stabilize the collective.

Theorem 3 ([11]). *Consider the collection of networked LTI systems in (16) and assume that at most C out of N systems are allowed to close their feedback*

[1]The discussion applies in the case of output feedback as well.

loops at any one time. For $i = 1, ..., N$, *let* $V_i^c(x_i) = x_i^T P_i x_i$, $P_i = P_i^T > 0$ *be Lyapunov functions for the closed-loop systems, satisfying* $(A_i^c)^T P_i + P_i A_i^c < \lambda_i P_i < 0$ *when communication is available (feedback loop closed) and* $(A_i^o)^T P_i + P_i A_i^o < \mu_i P_i$ *otherwise (for some* $\lambda_i < 0$, $\mu_i > 0$*). Then, for any* $T > 0$, *there exists a* T*-periodic communication sequence that stabilizes all* N *systems if*

$$\sum_{i=1}^{N} \frac{\mu_i}{\mu_i - \lambda_i} < C. \tag{17}$$

See also [4] for a condition based on rate-monotonic scheduling.

The parameters μ_i, λ_i in (17) are not unique but can be optimized to yield a less conservative bound. The optimization involves solving a set of bilinear matrix inequalities (see [11] for details).

For simplicity, assume from now on that $C = 1$, i.e., only one system can close its feedback loop at any one time, and consider the following communication policy [12].

Definition 2 (CLS-ϵ). *Let* $i_*(t)$ *denote the index of the system whose feedback loop is closed at time* t.

- *1. Let* t_0 *denote the current time. Set* $i_*(t_0) = \arg\max(\|x_i(t_0)\|)$.
- *2. When* $\|x_{i_*}(t)\| = \epsilon > 0$, *repeat from step 1.*

This policy, which seeks to "Contain the Largest State" (CLS-ϵ), can be viewed as the analog of the "Clear the Largest Buffer" policy, originally introduced in the study of distributed manufacturing systems [22]. CLS-ϵ chooses the system with the largest state and steers it near the origin, before selecting again. We note that such a policy cannot stabilize the collection; at best, it may guarantee that the systems are ϵ-*captured*, i.e., the $\|x_i\|$ will be arbitrarily close to ϵ as $t \to \infty$.

If the systems under consideration have *scalar dynamics*, we can obtain a necessary and sufficient condition for ϵ-capture.

Theorem 4 ([12]). *Consider the collection of networked LTI systems described in (16) with* $A_{s_i(t)} \in \{A_i^o, A_i^c\}$, $A_i^o > 0$, $A_i^c < 0$, *where at most* $C = 1$ *out of* N *systems are allowed to close their feedback loops at any one time and where the binary-valued* $s_i(t)$ *are determined by CLS-*ϵ *for any fixed* $\epsilon > 0$. *Then, all* $|x_i(t)|$ *will approach* ϵ *if and only if* [2] $\phi \stackrel{\triangle}{=} \sum_{i=0}^{N} \frac{A_i^o}{A_i^o - A_i^c} < 1$. *Furthermore, if* $\phi > 1$ *then there exists no stabilizing communication sequence.*

CLS-ϵ can also be used to drive the systems to the origin by gradually decreasing the value of ϵ. Under CLS-ϵ, the switching rate is not bounded. It is possible however to slightly modify the switching policy so that the switching rate is bounded above by $\frac{1}{\tau}$ [12]. The "minimum waiting" time $\tau > 0$ will

[2] For $k > 1$, replace the right-hand side of the inequality with k.

be the analog of the "setup time" in [22]. In that case (which will not be discussed here due to space constraints) the states will remain bounded.

If the systems of (16) are multivariable, then it is possible that CLS may fail to stabilize the collection but that there are other communication sequences that result in stability. In fact, there are well-known examples of switched systems for which there exists a stabilizing switching sequence, even when A^c and A^o are both unstable [3]. This suggests that, unlike the scalar case, there may be no necessary condition for stability based solely on the eigenvalues of the systems. However, sufficient conditions for stability or ϵ-capture can be obtained if we are willing to make switching decisions based not on the norms $\|x_i\|$ but rather on the exponential curves that bound the Lyapunov functions from Theorem 3, or on the Lyapunov functions V_i themselves [12]. In the latter case, one typically obtains a less conservative switching policy.

Theorem 5. *The collection of systems in (16) will be ϵ-captured under the interrupt-based communication policy obtained by replacing $\|x_i(t)\|$ by $V_i(x_i(t))$ in the CLS-ϵ algorithm, if $\phi = \sum_{i=1}^{N} \frac{\mu_i}{\mu_i - \lambda_i} < 1$, where λ_i and μ_i are obtained by solving Problem 1.*

In the latter case, the $V_i(x_i(t))$ are not pure exponentials and in fact may not be monotonic between switching times; therefore the state whose Lyapunov function is largest at a given switching time t may not always correspond to the system whose envelope function is largest at t.

We note that in Theorems 4 and 5, the CLS-ϵ policy must continuously attend to the states x_i in order to decide when a switching must take place. It is possible to modify matters so that making network access decisions requires only intermittent feedback (sampling of the $\|x_i\|$) or no feedback at all. In those cases, switching decisions are made based on a set of piecewise exponential curves that bound the Lyapunov functions V_i [12].

Fully coupled NCS

Consider now an NCS where the plant is the following controllable LTI system:

$$\dot{x} = Ax + Bu; \quad x(0) = x_0 \tag{18}$$

$$y = Cx; \quad x \in \mathbb{R}^n,\ u \in \mathbb{R}^m,\ y \in \mathbb{R}^p. \tag{19}$$

For now, assume a state feedback controller ($y(t) = x(t)$) and that only one of the n sensors can communicate with the controller while others must wait. At the input side of the plant, m actuators share a single input channel to communicate with the controller (what is discussed below can be easily extended to the multiple-access case).

This time, define a communication sequence as the continuous-time analog of Definition 1, namely $\sigma(t) : \mathbb{R} \mapsto \{0,1\}^M$, with $\|\sigma(t)\|^2 = N$, $\forall t$, so that a given output, say $x_i(t)$, is available to the controller only when $\sigma_i(t) = 1$;

otherwise, we assume (as in Section 3.1) that a *zero* value will be used by the controller for that sensor to generate the control signals, while the actual output $x_i(t)$ will be ignored due to its being unavailable [31].

The state x and its value $\bar{x}$ as seen from the controller (Fig. 1) are now related similarly to those in Section 3.1 so that under static feedback, $\bar{u}(t) = K \cdot \bar{x}(t)$, the closed-loop dynamics of the NCS are

$$\dot{x}(t) = (A + B \cdot \mathrm{diag}(\rho(t)) \cdot K \cdot \mathrm{diag}(\sigma(t)))\, x(t). \tag{20}$$

The medium access constraints are captured by cascading the plant with a pair of time-varying operators which are obtained directly from the input and output communication sequences. The stabilization problem can now be solved in a straightforward way, in contrast to the case when a ZOH was used between the communication medium and the plant.

By definition, $\rho(t)$ can only have m possible values and $\sigma(t)$ can only have n possible values. Hence the closed loop NCS (20) is essentially a switched system with $m \cdot n$ possible dynamics[3]:

$$\dot{x} = \mathcal{A}_{s(t)} x \tag{21}$$

where $s(t)$ defines a switching rule, $s(t) : \mathbb{R} \mapsto \{1, \ldots, m\} \times \{1, \ldots, n\}$ and $\mathcal{A}_{s(t)}$ takes values on the set $\{\mathcal{A}_{ij} : i = 1, \ldots, m;\ \ j = 1, \ldots, n\}$, where $\mathcal{A}_{ij}$ denotes the closed-loop dynamics when actuator i and sensor j are accessing the communication medium.

A stabilizing gain and communication policy can now be determined by the following algorithm [31], using a result [6] from switched systems:

- Choose $\Gamma \in \mathbb{R}^{p \times m}$ so that $A + B\Gamma$ is stable.
- Choose scalars $\alpha_{ij} > 0$, for $1 \leq i \leq m$, $1 \leq j \leq p$ so that $\sum \alpha_{ij} = 1$.
- Write $\Gamma = \sum_{i,j} \alpha_{ij} K_{ij}$ where K_{ij} are $p \times m$ basis matrices, whose (i, j) entry is the real variable k_{ij} and all other entries are zero.
- Notice that $A + B\Gamma = A + B \sum K_{ij} = \sum \mathcal{A}_{ij}$.
- The communication policy selects at any time t the sensor and actuator corresponding to the indices

$$i^*(t), j^*(t) = \arg\min_{i,j}\ x^T(t)(\mathcal{A}_{ij}^T P + P\mathcal{A}_{ij})x(t),$$

  where P is such that $(A + B\Gamma)^T P + P(A + B\Gamma) = -Q$, for some $Q = Q^T > 0$.
- The corresponding stabilizing feedback gain K is obtained by solving $\Gamma = \sum \alpha_{ij} K_{ij}$ for the K_{ij} and setting $K = \sum_{ij} K_{ij}$.

[3] When the communication medium has n_ρ $(1 < n_\rho < m)$ input channels and n_σ $(1 < n_\sigma < m)$ output channels, then $\rho(t)$ and $\sigma(t)$ will have $\binom{m}{n_\rho}$ and $\binom{n}{n_\sigma}$ possible values, respectively. The closed-loop system will then switch between $\binom{m}{n_\rho} \cdot \binom{n}{n_\sigma}$ possible dynamics.

This policy and gain will stabilize the NCS (20). It is easy to see why: given that $A + B\Gamma$ is stable by choice of Γ, there exist positive definite matrices P, Q such that $(A + B\Gamma)^T P + P(A + B\Gamma) = -Q < 0$, and for all $x(t) \neq 0$,

$$\sum_{i,j} \alpha_{ij} x^T(t)(\mathcal{A}_{ij}^T P + P\mathcal{A}_{ij})x(t) = -x^T(t)Qx(t) < 0.$$

Because $\alpha_{ij} > 0$ it follows that for all $x(t) \neq 0$ there always exist indices $i(x) \in \{1, \ldots, m\}$ and $j(x) \in \{1, \ldots, n\}$ such that $x^T(t)(\mathcal{A}_{i(x)j(x)}^T P + P\mathcal{A}_{i(x)j(x)})x(t) < 0$, which immediately gives us a choice of communication policy that keeps the Lyapunov function $V(t) = x^T(t)Px(t)$ always decreasing.

We note that for the same choice of $\mathcal{A}$, different choices of α_{ij}'s result in different values of the feedback gain K. A larger α_{ij} leads to a smaller k_{ij}. This fact gives us additional freedom in the design of K. By properly choosing α_{ij}'s we can make the controller K meet certain optimization or design criteria, or force the communication policy to pay more "attention" to certain sensors and actuators. A general communication policy might take the form [31]

Definition 3 (Weighted Fastest Decay (WFD)). *For all t, let $s(t) = (i(t), j(t))$ be determined by*

$$s(t) = \arg\min_{i,j} \alpha_{ij} \mathbf{x}^T(t)[\mathcal{A}_{ij}^T P + P\mathcal{A}_{ij}]\mathbf{x}(t), \tag{22}$$

where the coefficients α_{ij} act as weights associated with the dynamics $\mathcal{A}_{ij}$.

This class of policies is stabilizing, provided that K has been designed according to the algorithm given above. Modifications can also be made to ensure that the switching rate is bounded [31].

Definition 4 ([26]). *The system (21) is said to be quadratically stable if there exists a positive definite quadratic function $V(x) = \mathbf{x}^T P\mathbf{x}$, a positive number ϵ and a switching rule $s(t)$ such that $\frac{d}{dt}V(\mathbf{x}) < -\epsilon\mathbf{x}^T\mathbf{x}$ for all trajectories $\mathbf{x}$ of the system (21).*

Theorem 6 ([31]). *If $\mathcal{A}$ is stable, system (21) is quadratically stable under the switching rule WFD.*

The output feedback case can be handled by inserting a state observer between the communication medium and the feedback controller (see [30] for details in the discrete-time case).

3.3 The effects of transmission delays

We now discuss some of the effects of transmission delays on the stability of NCSs. We begin with the structure illustrated in Fig. 4, where sensor data are delayed by τ_s while actuator data are delayed by τ_a units of time. In practice,

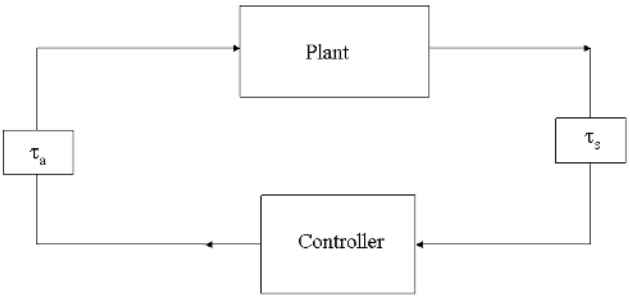


Fig. 4. NCS subject to transmission delays

these delays are induced not only by the finite speed at which data travel inside the communication medium, but also by the details of the communication protocol (e.g., single- or multiple-packet transmissions). For the remainder of this section we will assume single-packet transmission, i.e., all elements of $y(t)$ are transmitted together.

Let the continuous-time LTI plant evolve according to (18). The signal available to the controller is $x(kh-\tau_s)$. The (discrete-time) controller is given by

$$\bar{u}(t) = -Kx(kh-\tau_s); \quad k=0,1,2,..., \quad t \in [kh+\tau_s, (k+1)h+\tau_s)$$

and arrives at the plant τ_a seconds later. Thus, from the point of view of the plant, the total delay around a *constant-gain* feedback loop is the sum $\tau = \tau_s + \tau_a$, so that the plant is driven with a piecewise constant input which is obtained from the delayed sensor data:

$$u(t) = -Kx(kh-\tau); \quad t \in [kh+\tau, (k+1)h+\tau), \ k=0,1,2,... \tag{23}$$

The question for this type of NCS is the following.

Question 1. Given K such that $A+BK$ is stable, what is the maximum delay τ_{max} which can be tolerated before the NCS becomes unstable?

Constant delay

If τ is constant, then

Theorem 7 ([32]). *The NCS with constant delay (23) is stable if the eigenvalues of*

$$H = \left[\begin{array}{c|c} e^{Ah} & \int_0^h e^{A(h-s)}dsBK \\ \hline e^{A(h-\tau)} & -e^{A\tau}\left(\int_0^h e^{A(h-s)}ds - \int_0^\tau e^{A(h-s)}ds\right)BK \end{array}\right]$$

lie inside the unit disc.

Time-varying delays

If the loop delay τ_k varies from one transmission to the next, then the rate at which new inputs arrive at the plant is not fixed. The system can be analyzed by augmenting the state to include $x(kh)$ as well as all inputs that the plant receives in the interval $[kh, kh+\tau_k)$. The augmented state is $z(kh) = [x^T(kh), u^T((k-l)h), ..., u^T((k-1)h)]^T$, where the integer l is such that $(l-1)h < \tau_k < lh$ for all k. The stability of the (linear) dynamics of z is equivalent to the stability of the original NCS.

For example, if $l = 1$, then one must check the stability of the time-varying linear system [32] $z((k+1)h) = \Phi(k)z(kh)$ with $z(kh) = [x^T(kh), u^T((k-1)h)]^T$ and

$$\Phi(k) = \left[\begin{array}{c|c} e^{Ah} - \int_0^{h-\tau_k} e^{As} BdsK & \int_{h-\tau_k}^{h} e^{As} Bds \\ \hline -K & 0 \end{array}\right]. \tag{24}$$

As the discussion above suggests, the problem becomes significantly more complicated if τ_k varies in such a way that the integer l is not fixed.

We remark that it is sometimes possible to compensate for some of the effects of transmission delays. If the plant has the ability to "time-stamp" its sensor data before transmission, then the controller can use the time information to compensate for the delay τ_s, assuming that plant and controller have synchronized their clocks. The controller can then estimate the current state of the plant by propagating (according to the dynamics (18)) the state data it receives from the sensors, for an amount of time equal to that of the sensor-to-controller delay. If the controller-to-actuator delay τ_a is constant, then the controller can also compensate for the amount of time its data will take to reach the plant. For further details, including the construction of a delay-compensating state observer, see [32] and references therein. See also [16, 21] for discussions of NCS stability under random delays.

3.4 The effects of unreliable communication links

We now turn our attention to the possibility that the communication between plant and controller is unreliable, in the sense that sensor/actuator data may fail to reach their destination. This situation, where data packets are "dropped," can arise because of network congestion, unreliable hardware, or because of the transmission protocol used (the transmission control protocol (TCP) and user datagram protocol (UDP) are two well-known examples).

Consider an NCS, where the connections from controller to plant and plant to controller (referred to as *uplink* and *downlink*, respectively) are unreliable, in the sense that transmitted data may occasionally fail to reach its destination. To make things precise, let the plant be described by the discrete-time LTI system

$$x(k+1) = Ax(k) + \alpha(k)Bu(k), \qquad (25)$$
$$y(k) = \beta(k)x(k),$$

where $\alpha(k), \beta(k) \in \{0, 1\}$ indicate whether at time k the control (measurement) signal reaches the plant (controller) or not. The assumption here is that the vectors u and y are sent in single-packet transmissions, and that the sequences $\{\alpha(k)\}$, $\{\beta(k)\}$ are i.i.d. Bernoulli, with $Pr[\alpha(k) = 0] = \alpha$ and $Pr[\beta(k) = 0] = \beta$ being the link failure probabilities.

One can then pose the following problem.

Problem 3. For the system of (25) and given the link failure rates α, β, find a control policy that minimizes

$$J = \mathcal{E}\Big\{ \sum_{k=0}^{\infty} x(k)^T Q x(k) + \alpha(k) u(k)^T R u(k) \Big\},$$

where $\mathcal{E}\{\cdot\}$ denotes expected value.

This LQR problem (and its finite-horizon version) are discussed in [13]. A related problem is the following.

Problem 4. For the system (25), what are the maximum link failure rates α, β for which a stabilizing controller exists?

In [13] it is shown that the controller that minimizes J is given by a feedback law similar to that for the standard LQR problem:

$$u^*(k) = G(k)\hat{x}(k); \quad \hat{x}(k) \stackrel{\triangle}{=} \mathcal{E}\{x(k)\}, \qquad (26)$$

where $\hat{x}(k)$ is an estimate of the state x obtained by

$$\hat{x}(k) = \begin{cases} A\hat{x}(k) + \alpha(k-1)Bu(k-1), & \beta(k) = 0 \\ x(k), & \beta(k) = 1 \end{cases}, \qquad (27)$$

$G(k) = -(R + B^T K(k+1)B)^{-1} B^T K(k+1)A$, and $K(k)$ is determined by the following recursive matrix equations:

$$P(k) = (1-\alpha)A^T K(k+1)B(R + B^T K(k+1)B)^{-1} B^T K(k+1)A \qquad (28)$$
$$K(k) = A^T K(k+1)A - P(k) + Q. \qquad (29)$$

The solution to the last set of equations as well as the answer to Problem 4 depend strongly on whether the communication protocol includes "acknowledgment" (ACK) packets that allow the sender to know whether its transmission was received or not. If all receptions are acknowledged and ACK packets are always received by the transmitter, then the separation principle [7] holds, and the controller and estimator can be designed separately. If the protocol does not support acknowledgment (e.g., UDP), then the controller does not

know the "state of the channel," i.e., whether $\alpha(k)$ was 0 or 1 and thus has no way of knowing whether the sensor output it receives at time $k+1$ includes the effect of $u(k)$.

In the case of the infinite-horizon LQR problem with acknowledgments, the stabilizing controller and maximum link failure rate are given by the following theorem.

Theorem 8 ([13]). *Let B be square and full rank, and let $(A, Q^{1/2})$ be observable. Suppose*

$$\max_i |\lambda_i(A)| < \min\left\{\frac{1}{\sqrt{\alpha}}, \frac{1}{\beta}\right\}.$$

Then (i) $K(k)$ converges to the positive definite solution of

$$K = A^T K A + Q - (1-\alpha) A^T K B (R + B^T K B)^{-1} B^T K A$$

and (ii) the closed-loop system is stable.

Without ACK packets the estimator is again given by (27). However, the separation principle does not hold and the maximum tolerable link failure rate is slightly reduced, as given the following theorem.

Theorem 9 ([13]). *Let B be square and full rank, and let $(A, Q^{1/2})$ be observable. Suppose*

$$\max_i |\lambda_i(A)| < \min\left\{\sqrt{\frac{1+\alpha\beta}{\alpha+\alpha\beta}}, \frac{1}{\beta}\right\}.$$

Then (i) there exist $K > 0$, $P > 0$, such that for $P(0) = 0$ and all $K(0) > 0$, the Riccati equations (29) converge to the positive definite solutions of

$$P = (1-\alpha) A^T K B (R + B^T K B)^{-1} B^T K A \tag{30}$$

$$K = A^T K A - P + Q \tag{31}$$

and (ii) the closed-loop system is stable.

The case where the system is subject to actuator noise and transmissions are multiple-packet is discussed in [1]. In that work acknowledgment packets can also be "dropped," so that the separation principle does not hold. Thus one has a system whose dynamics switch randomly between eight possible dynamics, depending on whether the transmitted data (from either controller or plant) arrived at its destination, and whether an acknowledgment failed to arrive back to the sender. One can design a suboptimal controller/estimator pair (by insisting on separation) and arrive at a set of necessary and sufficient linear matrix inequality(LMI)-based conditions that relate the stability of the closed-loop NCS (under the proposed controller/estimator structure) to the link failure rates.

For the special case where only the downlink is subject to unreliable communication under single-packet transmission, and the controller $u(k) = K\hat{x}(k)$ uses ZOH,

$$\hat{x}(k) = \begin{cases} x(k) & \text{if } \beta(k) = 1, \\ \hat{x}(k-1) & \text{if } \beta(k) = 0 \end{cases} \tag{32}$$

a bound for the maximum allowable link failure rate is given in [32].

Theorem 10 ([32]). *Consider the system of (25) where $\alpha(k) = 1$ for all k, i.e., only the downlink is subject to failures, with rate $(1 - r)$, $0 < r \leq 1$. If the controller K is such that $A + BK$ is stable, then the closed-loop system is exponentially stable for all*

$$\frac{1}{1 - \log\left(\lambda^2_{\max}(A + BK)\right) / \log\left(\lambda^2_{\max}(A)\right)} < r \leq 1.$$

3.5 Communication sequences: Beyond stability-related problems

In addition to the stabilization problems discussed in the previous sections, communication sequences have been used to capture communication constraints in problems related to tracking, optimal control and robust control. For example, the work in [23] discusses LQR problems with communication constraints, and [8] addresses least-squares output tracking for NCS. As we have mentioned, the problem of finding optimal communication sequences is typically a difficult one. Interesting heuristics that attempt to find near-optimal communication sequences are explored in [18] (H_2 and H_∞ control for NCS) and [17] (optimizing communication in LQR problems).

4 A Complementary Viewpoint: Control with Limited Bit Rate

Up to now, we have concentrated on time-division based models for capturing communication constraints and have treated the communication channel as being able to transmit signals with infinite precision. This assumption works well when channel throughput is high enough so that performance is not significantly affected by quantization errors. Aside from the fact that realistic channels can only accommodate a finite number of bits per second, re-examining limited communication control where the limited resource is *bits* as opposed to time can yield valuable insights as to how one could design controllers for NCSs, and what data rates are required for adequate performance. The rest of this section reviews some of the fundamentals when the feedback loop is closed via digital channels which are subject to data rate limitations.

4.1 State estimation and stabilization with limited bit rate

Fig. 5 illustrates an NCS whose feedback loop is closed over a digital channel. The channel can support a maximum rate of R bits per second (it takes $\delta = 1/R$ seconds to transmit a single bit). Assume that the plant is continuous-

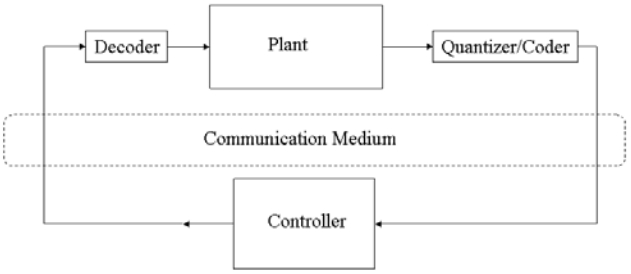


Fig. 5. A feedback loop which is closed over a communication channel with limited bit rate. The controller acts on coded versions of the sensor data and produces inputs that must be decoded before being applied to the plant

time LTI (18). Digital communication means that sensor and actuator data must be sampled with finite precision (quantized) and coded in a string of bits (or more generally, symbols from a D-ary alphabet). The coded observation is sent to the controller, which decodes it and computes a new plant input, which is again coded before being transmitted to the plant. There are various choices with respect to the coding scheme used, including fixed- and variable-length codes. A particularly useful subset of the latter category are *prefix codes* [5], which allow the receiver to immediately recognize the end of a code word.

Question 2. What is a necessary (sufficient) bit rate to guarantee the existence of a coding scheme and estimator for the state x?

Question 3. What is a necessary (sufficient) bit rate to guarantee the existence of a coder, decoder and controller that stabilize the NCS?

Estimation

We begin by addressing Question 2. A "good" estimate of the state x seems necessary in order for any controller to be effective. Under the assumptions given in the previous section, the time required to transmit a measurement of the state grows linearly with the number of bits sent. Therefore, precision becomes counterproductive after a certain point: if one chooses to describe a sensor reading too precisely, the controller will receive the digital description later and the data will be outdated.

To illustrate this trade-off, assume that fixed-length coding is used to send data from the plant to the estimator, and that the plant is discrete time, with no process or measurement noise:

$$x(k+1) = f_k(x(k)), \quad x(k) \in \mathbb{R}^n, \quad x(0) \in S_0 \subset \mathbb{R}^n, \quad k = 0, 1, 2, \ldots . \quad (33)$$

The initial state $x(0)$ is assumed to be drawn from a probability density with compact support (e.g., a compact set S_0) and the functions f_k are Lipschitz:

$$\|f_k(x) - f_k(y)\| \leq M_k \|x - y\| \quad \forall x, y \in \mathbb{R}^n.$$

At each time step k, an estimator receives R bits of information on the state $x(k)$ (propagation delays are ignored) and must produce an estimate $\hat{x}(k+1)$, based on all *past transmissions*. One way to do this is to begin by partitioning S_0 into 2^R disjoint regions and transmit the index of the region that contains $x(0)$. The estimator can then set the auxiliary variable $z(0)$ to be any point in that region (e.g., its centroid) and then propagate $z(0)$ ahead according to the dynamics (33) to generate the current estimate, i.e., at time k, set $\hat{x}(k+1) = f_k(z(k))$. The key point is that if the error $e(k) = x(k) - \hat{x}(k)$ does not grow too rapidly with k (equivalently, if the Lipschitz constants M_k are not too large), the estimator can know with certainty that the state $x(k+1)$ is contained in a subset $S(k+1)$ whose size (in terms of diameter or Lesbegue measure) is smaller than that of $S(k)$. The new $S(k+1)$ is then repartitioned in 2^R subsets and the procedure is repeated, further "narrowing down" the expected value of the error. The following is a restatement of the main result in [19].

Theorem 11 ([19]). *Let the distribution of $x(0)$ have compact support. Then, there exist a coder and estimator (based on the successive partitioning idea described above) such that*

$$\mathcal{E}\|x(k+1) - \hat{x}(k+1)\|^2 \leq \phi^2 \left(\frac{\prod_0^k M_i}{2^{Rk/n}} \right),$$

where ϕ is a constant that depends on the state dimension n, the bit rate R and the size of the compact set that contains the initial state.

The last inequality gives a sufficient condition for the estimation error to converge to zero. If the plant is linear (35), then the same condition specializes to

$$R > n \log_2(\max_i \lambda_i(A)).$$

The work in [19] includes extensions of the last theorem in the case where the distribution of $x(0)$ is not compact. For additional insights into the problem of state estimation under process and measurement noise, including an explicit consideration of transmission delays, see [27].

Stabilization: Explicit consideration of transmission delays

We now turn to the problem of stabilizing the NCS (Fig. 5) under the bit rate constraints discussed in the previous section. We will assume for the moment that controls are applied for arbitrarily short time intervals. The finite delay between when y (or x) is measured and u is applied means that the system is essentially uncontrolled for some time initially and cannot be asymptotically stabilized unless the initial state and start time are known precisely. However, one can ask for a slightly weaker type of stability.

Definition 5. *An NCS is containable if for any ball N around the origin there exists an open neighborhood of the origin M and coding and feedback control laws such that any trajectory started in M remains in N for all time.*

Question 3 can be answered (in the context of containability) by assuming first that $x(0)$ lies in some Lesbegue-measurable set S_0. Suppose the plant is continuous-time LTI (18). If the plant is unstable, then any uncertainty in one's knowledge of $x(0)$ will be "amplified" as time goes on. On the other hand, the coder must balance speed with precision (i.e., giving an "answer" quickly versus transmitting more bits) when providing information on x.

We begin by asking how many bits it takes to "narrow down" the set that contains x over consecutive transmissions, given the bit rate of $1/\delta$ [28]. After applying a control u to the plant over some interval $[0, k\delta]$, and in the absence of any observations, the state must lie somewhere in the set

$$S_1(S_0, u, T) = e^{At} S_0 + g(u),$$

where $g(u)$ is some vector in $\mathbb{R}^n$ and $e^{At} S \triangleq \{e^{At} x : x \in S\}$.

If we let $\mu(S)$ denote the n-dimensional "volume" (more precisely the Lesbegue measure) of the set S, we have

$$\mu(K_1(S_0, u, T)) = \det(e^{At}) \mu(S_0) = e^{tr(A)t} \mu(S_0).$$

As before, consider decomposing the set S_0 into $S_0 = \cup_{i=0}^{N} K_i$ where the subsets K_i are such that all elements of K_i correspond to a unique code word c_i. The coder checks to see which of the K_i contains the state and transmits the corresponding code word (using c_i bits) to the controller, which in turn sends d_i bits to the plant. The c_i, d_i are assumed to be prefix codes. Then, the set that contains the state at the end of the first transmission satisfies

$$\mu(S_1) = e^{tr(A)(c_i + d_i)\delta} \mu(K_i).$$

If the system is to be containable, then $\mu(S_1) < \sum \mu(K_i)$ because $S_1 \subset S_0 = \cup_i K_i$. Summing over all K_i leads to

$$\sum_{0}^{N} \frac{1}{e^{\delta(c_i + d_i) trA}} \geq 1 \tag{34}$$

as a necessary condition for containability. Moreover, if the same set of code words is used for both observation and control, then the following can be shown [28].

Theorem 12 ([28]). *The NCS is containable only if* $e^{2tr(A)\delta} \leq D$.

In the case where y, x and u all have the same dimension n, one can obtain a sufficient condition as well.

Theorem 13 ([28]). *If* (A, B) *is controllable and* C *is invertible, then the NCS (with binary code words) is containable if* $\max_{\|x\|_\infty} \|e^{\delta A}x\|_\infty^{2^n+1} < 2$.

Stabilization with "instantaneous" transmissions

The situation discussed in the last section makes a very useful connection between the bit rate supported by the channel and the size of the alphabet in which data is sent. It also shows explicitly that containability will be violated if one chooses to be too precise about measurements (34) and that—as expected—increasing the size of the alphabet improves the bound because it increases the amount of information carried by each bit around the loop. A similar but slightly "tighter" condition can be obtained for the discrete-time counterpart of (18) [20]:

$$x(k+1) = Ax(k) + Bu(k); \quad x(0) \in K_0 \subset \mathbb{R}^n \tag{35}$$

$$y(k) = Cx(k), \tag{36}$$

where we assume that the plant and controller are co-located, i.e., there is no need for coding/decoding actuator data and only sensor measurements must be transmitted through a digital channel at a rate of R bits per time step.

Theorem 14 ([20]). *Assume that the system (35) is reachable and observable and that its initial state* $x(0)$ *is random, with a distribution which is absolutely continuous with respect to the Lesbegue measure on* $\mathbb{R}^n$ *and has finite* $(r+\epsilon)$*-th absolute moment* $\mathcal{E}\|x_0\|^{r+\epsilon} < \infty$ *for some* $r, \epsilon > 0$. *Then, for a given data rate* R *(bits per step* k*), a coder/controller that exponentially stabilizes the NCS with rate* ρ*, i.e.,*

$$\lim_{k\to\infty} \rho^{-kr}\mathcal{E}\|x(k)\|^r = 0,$$

exists if and only if

$$R > \sum_{|\lambda_i|\geq\rho} \log_2 \left|\frac{\lambda_i(A)}{\rho}\right|, \tag{37}$$

where $\lambda_i(A)$ *are the eigenvalues of* A.

The same condition on R is thoroughly explored in [25] in the context of observability, controllability and exponential stability of the NCS (35). See

also [20, 29] for additional discussions of (37), including details of how various choices of coding/quantization schemes affect the bounds on the bit rate necessary for stability.

If $C = I$, the condition (37) is necessary and sufficient for the existence of a stabilizing encoder/decoder/controller. In the case where the system (35) is subject to bounded input noise,

$$x(k+1) = Ax(k) + Bu(k) + w(k); \quad \|w(k)\| < M \tag{38}$$

$$y(k) = x(k), \tag{39}$$

then (37) is necessary and sufficient for the existence of a coder, decoder and controller for which the estimation error $\limsup_{k\to\infty} \|x(k) - \hat{x}(k)\|$ is bounded, where $\hat{x}(k)$ is the output of the decoder. Here, the existence of the encoder is asserted over all encoders that have access to all past observations and controls; both encoder and decoder are assumed to have knowledge of the dynamics of the plant as well as one another.

5 Beyond this Introduction

This chapter explored some of the important results in the area of control with limited communication, focusing mainly on stabilization and estimation problems. Our goal was to give the reader a "flavor" of how communication constraints enter into the solution of control problems and to describe some of the available tools for designing (jointly when possible) effective controllers and communication policies. We explored various types of communication constraints, including transmission delays, unreliable communication links, and what could be termed "bottlenecks." The latter were due either to the limited number of channels available for controller-plant communication or to the limited throughput of a single channel shared by all sensors or actuators.

The area of NCSs is at the interface of several "core" fields within systems and control. Some of the results discussed here were built on previous contributions in switched and hybrid systems (see related chapters in this book), as well as results from more "mature" areas such as periodic systems and information theory. An excellent collection of NCS-related references can be found online at `http://home.cwru.edu/ncs/allpubs.htm`. Other areas that may offer the reader additional useful viewpoints on the interplay of control and communication, but were not mentioned in this chapter, include multirate systems, scheduling, systems with delays and quantized control.

References

1. B. Azimi-Sadjadi. Stability of networked control systems in the presence of packet losses. In *Proceedings, 42nd IEEE Conference on Decision and Control*, pages 676–81, Dec. 2003.

2. V. Blondell and J. Tsitsiklis. NP hardness of some linear control design problems. *SIAM Journal on Control and Optimization*, 35:2118–2127, 1997.
3. M. S. Branicky. Multiple Lyapunov functions and other analysis tools for switched and hybrid systems. *IEEE Trans. on Automatic Control*, 43(4):475–82, April 1998.
4. Michael S. Branicky, Stephen M. Phillips, and Wei Zhang. Scheduling and feedback co-design for networked control systems. In *Proceedings of the 38th Conference on Decision and Control*, pages 1211–1217, December 2002.
5. T. M. Cover and J. A. Thomas. *Elements of Information Theory*. Wiley, New York, 1991.
6. Eric Feron. Quadratic stabilization of switched systems via state and output feedback. Technical Report CICS-P-468, Center for intelligent control systems, MIT, Cambridge MA, 1996.
7. B. Friedland. *Control System Design: An Introduction to State-Space Methods*. McGraw-Hill, New York, 1986.
8. D. Hristu. Generalized inverses for finite-horizon tracking. In *Proceedings of the 38th IEEE Conference on Decision and Control*, volume 2, pages 1397–402, 1999.
9. D. Hristu. Stabilization of LTI systems with communication constraints. In *American Control Conference*, volume 4, pages 2342–6, June 2000.
10. D. Hristu and K. Morgansen. Limited communication control. *Systems and Control Letters*, 37(4):193–205, July 1999.
11. D. Hristu-Varsakelis. Feedback control systems as users of a shared network: Communication sequences that guarantee stability. In *IEEE Conference on Decision and Control*, pages 3631–6, Dec. 2001.
12. D. Hristu-Varsakelis and P. R. Kumar. Interrupt-based feedback control over a shared communication medium. In *Proc. of the IEEE Conference on Decision and Control*, pages 3223–8, Dec. 2002.
13. O. C. Imer, S. Yuksel, and T. Basar. Optimal control of dynamical systems over unreliable communication links. In *Proc. of the 6th IFAC Symposium on Nonlinear Control Systems (NOLCOS)*, Sept. 2004.
14. H. Ishii and B. Francis. Stabilization with control networks. *Automatica*, 38:1745–51, 2002.
15. P. T. Kabamba. Monodromy eigenvalue assignment in linear periodic systems. *IEEE Trans. on Automatic Control*, 31(10):950–2, Oct. 1986.
16. B. Lincoln. Jitter compensation in digital control systems. In *Proceedings of the American Control Conference*, 2002.
17. B. Lincoln and B. Bernhardsson. Efficient pruning of search trees in LQR control of switched linear systems. In *Proceedings of the 39th IEEE Conference on Decision and Control*, volume 2, pages 1828–33, Dec. 2000.
18. L. Lu., X. Lihua, and M. Fu. Optimal control of networked systems with limited communication: A combined heuristic search and convex optimization approach. In *Proceedings of the 42nd IEEE Conference on Decision and Control*, pages 1194–9, Dec. 2003.
19. G. N. Nair and R. J. Evans. State estimation via a capacity-limited communication channel. In *Prof. of the 36th IEEE Conference on Decision and Control*, pages 866–71, Dec. 1997.
20. G. N. Nair and R. J. Evans. Exponential stabilizability of finite-dimensional linear systems with limited data rates. *Automatica*, 39:585–93, 2003.

21. J. Nilsson, B. Bernhardsson, and B. Wittenmark. Stochastic analysis and control of real-time systems with random time delays. *Automatica*, 23(1):57–64, 1998.
22. J. Perkins and P. R. Kumar. Stable distributed, real-time scheduling of flexible manufacturing/assembly/disassembly systems. *IEEE Trans. on Automatic Control*, 34(2):139–48, Feb. 1989.
23. H. Rehbinder and M. Sanfridson. Scheduling of a limited communication channel for optimal control. In *Proceedings of the 39th IEEE Conference on Decision and Control*, volume 1, pages 1011–16, Dec. 2000.
24. W. J. Rugh. *Linear Systems Theory*. Prentice-Hall, Englewood Cliffs, NJ, 1996.
25. S. Tatikonda and S. Mitter. Control under communication constraints. *IEEE Trans. Automatic Control*, 49(7):1056–68, Jul. 2004.
26. M. Wicks, P. Peleties, and R. DeCarlo. Switched controller synthesis for the quadratic stabilization of a pair of unstable linear systems. *European Journal of Control*, 4:140–7, 1998.
27. W. S. Wong and R. W. Brockett. Systems with finite communication bandwidth constraints–Part I: State estimation problems. *IEEE Transactions on Automatic Control*, 42(9):1294–1299, 1997.
28. W. S. Wong and R. W. Brockett. Systems with finite bandwidth constraints–Part II: Stabilization with limited information feedback. *IEEE Transactions on Automatic Control*, 42(5):1049–1052, 1999.
29. S. Yuksel and T. Basar. Quantization and coding for decentralized LTI systems. In *Proceedings of IEEE Conference on Decision and Control*, pages 2847–52, Dec. 2003.
30. L. Zhang and D. Hristu-Varsakelis. Stabilization of networked control systems: Communication and controller co-design. In *subm. to the IFAC World Congress*, 2005.
31. L. Zhang and D. Hristu-Varsakelis. Stabilization of networked control systems under feedback-based communication. In *subm. to the American Control Conference*, 2005.
32. W. Zhang, M. S. Branicky, and S. M. Phillips. Stability of networked control systems. *IEEE Control Systems Magazine*, 21(1):84–99, Feb. 2001.

Networked Control Systems: A Model-Based Approach

Luis A. Montestruque and Panos J. Antsaklis

University of Notre Dame, Notre Dame, IN 46556, U.S.A.
{lmontest, pantsakl}@nd.edu

1 Introduction

A networked control system (NCS) is a control system in which a data network is used as a feedback medium. NCS is an important research area; see for example [16] and [15, 18, 19]. The use of networks as media to interconnect the different components in an industrial control system is rapidly increasing, although the use of an NCS poses some challenges. One of the main problems to be addressed when considering an NCS is the size of the bandwidth required by each subsystem. It is clear that the reduction of bandwidth necessitated by the communication network in an NCS is a major concern. This can perhaps be addressed by two methods: the first method is to minimize the transfer of information between the sensor and the controller/actuator; the second method is to compress or reduce the size of the data transferred at each transaction. Since shared characteristics among popular industrial networks are a small transport time and a big overhead, using less bits per packet has a small impact on the overall bit rate. *So reducing the rate at which packets are transmitted brings better benefits than data compression in terms of the bit rate used.* In this chapter, we consider the problem of reducing the packet rate of an NCS using a novel approach called model-based NCS (MB-NCS). The MB-NCS architecture makes explicit use of knowledge about the plant dynamics to enhance the performance of a system. The MB-NCS was introduced in [11].

In Section 2 the basic MB-NCS setup is introduced for continuous plants. Stability of the MB-NCS with no quantization and periodic transmissions is considered. In Section 3 a performance measure is introduced for the previously presented MB-NCS. The stability of the MB-NCS with time-varying and stochastic transmission times is studied in Section 4. Quantization schemes are considered in Section 5. Finally, stability for a class of nonlinear systems is studied in Section 6.

2 Stability of Continuous Linear MB-NCS with Constant Update Times

2.1 State feedback MB-NCS

We consider the control of a continuous linear plant where the state sensor is connected to a linear controller/actuator via a network. In this case, the controller uses an explicit model of the plant that approximates the plant dynamics and makes possible the stabilization of the plant even under slow network conditions.

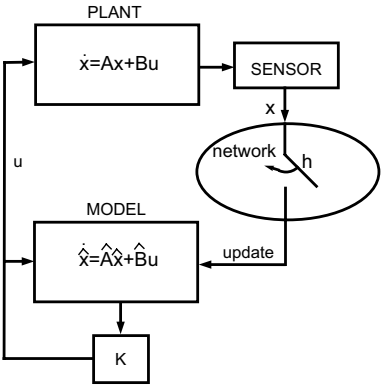


Fig. 1. Proposed configuration of networked control system (NCS)

We will concentrate on characterizing the transfer time between the sensor and the controller/actuator, which is the time between information exchanges. The plant model is used at the controller/actuator side to recreate the plant behavior so that the sensor can delay sending data since the model can provide an approximation of the plant dynamics. *The main idea is to perform the feedback by updating the model's state using the actual state of the plant that is provided by the sensor. The rest of the time the control action is based on a plant model that is incorporated in the controller/actuator and is running open loop for a period of h seconds.* The control architecture is shown in Fig. 1.

Our approach is novel in that it incorporates a model of the plant, the state of which is updated at discrete intervals by the plant's state. We present a necessary and sufficient condition for stability that results in a maximum transfer time. We make use of standard stability definition such as the ones found in [1].

If information for all the states is available, then the sensors can send this information through the network to update the model's vector state. For our analysis we will assume that the compensated model is stable and that the transportation delay is negligible. We will assume that the frequency at which

the network updates the state in the controller is constant. The idea is to find the smallest frequency at which the network must update the state in the controller, that is, an upper bound for h, the update time. Usual assumptions in the literature include requiring a stable plant, or in the case of a discrete controller, a smaller update time than the sampling time. Here we do not make any of these assumptions and the plant may be unstable.

Consider the control system of Fig. 1 where the plant is given by $\dot{x} = Ax + Bu$, the plant model by $\dot{\hat{x}} = \hat{A}\hat{x} + \hat{B}u$, and the controller by $u = K\hat{x}$. The state error is defined as $e = x - \hat{x}$, and it represents the difference between the plant state and the model state. The modeling error matrices $\tilde{A} = A - \hat{A}$ and $\tilde{B} = B - \hat{B}$ represent the difference between the plant and the model. We will now express the combined state $z(t) = \begin{bmatrix} x(t)^T & e(t)^T \end{bmatrix}^T$ in terms of the initial condition $x(t_0)$. A necessary and sufficient condition for stability of the state feedback MB-NCS will then be presented. Define $\Lambda = \begin{bmatrix} A + BK & -BK \\ \tilde{A} + \tilde{B}K & \hat{A} - \tilde{B}K \end{bmatrix}$.

Proposition 1 *[13] The state feedback MB-NCS with initial conditions* $z(t_0) = \begin{bmatrix} x(t_0)^T & 0^T \end{bmatrix}^T = z_0$ *has the following response:*

$$z(t) = e^{\Lambda(t-t_k)} \left(\begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} e^{\Lambda h} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} \right)^k z_0,$$
$$\text{for } t \in [t_k,\, t_{k+1}), \text{ with } t_{k+1} - t_k = h.$$

Theorem 1 *[13] The state feedback MB-NCS is globally exponentially stable around the solution* $z = \begin{bmatrix} x^T & e^T \end{bmatrix}^T = 0$ *if and only if the eigenvalues of* $M = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} e^{\Lambda h} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}$ *are strictly inside the unit circle.*

One can gain insight into the closed-loop system behavior by noticing that the non-zero eigenvalues of $M = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} e^{\Lambda h} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}$ are exactly the eigenvalues of another matrix:

$$N = e^{(\hat{A}+\hat{B}K)h} + e^{Ah} \int_0^h e^{-A\tau} (\tilde{A} + \tilde{B}K) e^{(\hat{A}+\hat{B}K)\tau} d\tau. \tag{1}$$

This can be shown either directly or by using the Laplace transform. The second method involves replacing the update time variable h by the time variable t, transforming the expression using the Laplace transform. The transformed expression can then be easily manipulated. After isolating the upper left block matrix, the inverse Laplace transform can be used to return to the time domain [10].

First we observe that the eigenvalues of the compensated model appear in the first term of N. We can see the term $\Delta = e^{Ah} \int_0^h e^{-A\tau} (\tilde{A} + \tilde{B}K) e^{(\hat{A}+\hat{B}K)\tau} d\tau$ as a perturbation on the desired eigenvalues, that is, the

eigenvalues of the compensated model. Even if the eigenvalues of the original plant were unstable, the perturbation Δ can be made small enough by having h and $\tilde{A}+\tilde{B}K$ small and thus minimizing their impact over the eigenvalues of the compensated plant. We also observe that if the update time h is driven to zero, then $\Delta = 0$. Also, it is possible to make $\Delta = 0$ by having a plant model that is exact.

Example 1 *Consider the following unstable plant (double integrator):* $A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. *We will use the state feedback controller given by* $u = Kx$ *with* $K = \begin{bmatrix} -1 & -2 \end{bmatrix}$. *Usually it is assumed that the actuator/controller will hold the last value received from the sensor until the next time the sensor transmits and a packet is received. Under this assumption the controller/actuator's model acts as a zero-order hold when updated. We will first analyze this situation. To do so, we will transform the plant model so that it holds the last state update presented to it by the network. This is given by* $\dot{\hat{x}} = \hat{A}\hat{x} + \hat{B}u$, *with* $\hat{A} = \begin{bmatrix} 0 & 0 \\ 0 & 0 \end{bmatrix}$ *and* $\hat{B} = \begin{bmatrix} 0 & 0 \end{bmatrix}^T$. *Now we need to search for the biggest* h *such that* $\begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} e^{\Lambda h} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}$ *has its eigenvalues inside the unit circle. To do so we plotted the maximum eigenvalue magnitude versus the update time. The plot is shown in Fig. 2.*

From Fig. 2 we see that the condition for stability is to have $h < 1$ *second. In fact, the test matrix* M *will have one eigenvalue with magnitude 1 for* $h = 1$ *second and the system will be marginally stable. If we use the results by [19] we would obtain that, in order to stabilize the system, we would need to have* $h < 2.1304 \times 10^{-4}$ *seconds, which is very conservative. Simulations of the system with update times of* 2.1304×10^{-4} *and* 0.5 *seconds are shown in Fig. 3. Note that the plant was initialized with an initial condition of* $\begin{bmatrix} 1 & 1 \end{bmatrix}^T$.

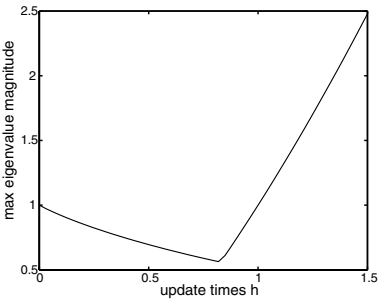


Fig. 2. Maximum eigenvalue magnitude of the test matrix M versus the update time h

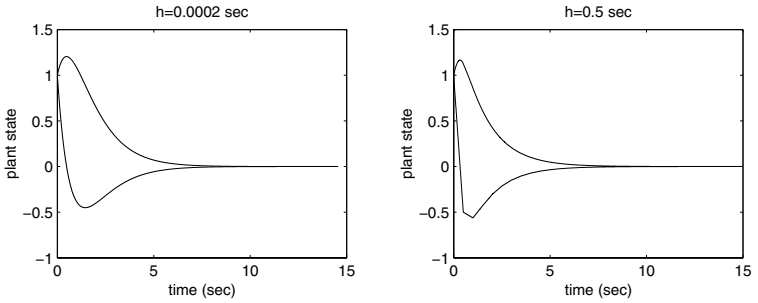


Fig. 3. System response for $h = 2.1304 \times 10^{-4}$ and $h = 0.5$ sec.

It is clear that the performance obtained with $h = 0.5$ second is not too different from the one obtained with $h < 2.1304 \times 10^{-4}$ seconds, but the difference in the amount of bandwidth used is large. If we were to use Ethernet that has a minimum message size of 72bytes (including preamble bits and start of delimiter fields), the data rate would be 2.7Mbits/sec for the case of $h < 2.1304 \times 10^{-4}$ seconds, and 1.2Kbits/sec for the case of $h = 0.5$ second.

Example 2 *In real applications uncertainties can frequently be expressed as tolerances over the different measured parameter values of the plant. This can be mapped into structured or parametric uncertainties on the state space matrices. Next an example is given on how Theorem 1 can be applied if two entries on the A matrix of the model can vary within a certain interval.*

$$\begin{aligned}
&\textit{model:} && \hat{A} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \hat{B} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}; \\
&\textit{plant:} && A = \begin{bmatrix} 0 & 1+\tilde{a}_{12} \\ 0+\tilde{a}_{21} & 0 \end{bmatrix}, B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}; \\
&\textit{with} && \tilde{a}_{12} = [-0.5, 0.5], \tilde{a}_{21} = [-0.5, 0.5] \\
&\textit{controller:} && K = \begin{bmatrix} -1 & -2 \end{bmatrix}.
\end{aligned}$$

The system will now be tested for an update time of $h = 2.5$ seconds. The following contour plot in Fig. 4 represents the magnitude of the maximum eigenvalue of the test matrix M as a function of the (1,2) and (2,1) entries of A parameters a_{12} and a_{21}. Here the contour of height equal to one is the relevant one to stability. It is easy to isolate the stable and unstable regions in the uncertainty parameter plane. The stable region is between the lines labeled as 1.

2.2 Output feedback MB-NCS

We have been considering plants where the full state vector is available at the output. We now extend our approach to include plants where the state is not

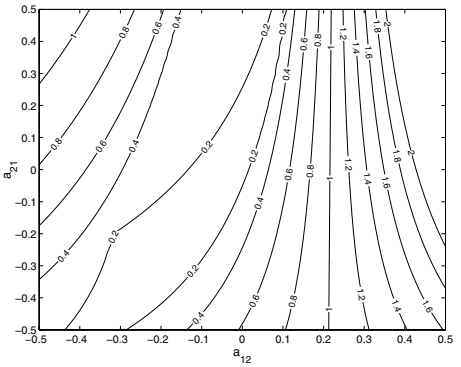


Fig. 4. Contour plot of maximum eigenvalue magnitude versus model error

directly measurable. In this case, in order to obtain an estimate of the plant state vector, a state observer is used. It is assumed that the state observer is collocated with the sensor. Again, we use the plant model, $\dot{\hat{x}} = \hat{A}\hat{x} + \hat{B}u$, to design the state observer. The observer uses the plant output and generates a copy of the plant input applied by the controller. See Fig. 2.2.

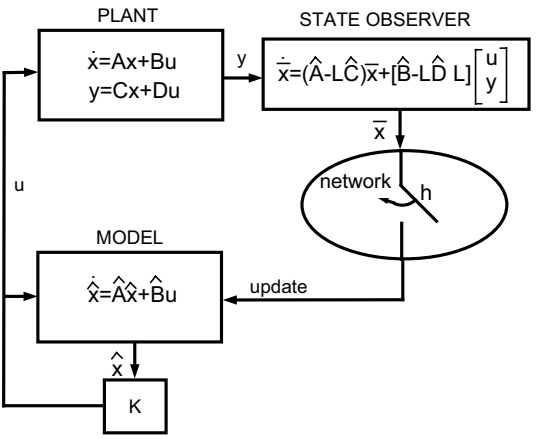


Fig. 5. Proposed configuration of an output feedback MB-NCS

Having the sensor carry the computational load of an observer is justified by the fact that typically sensors that can be connected to a network contain an embedded processor. This processor is usually in charge of performing the sampling and filtering and implementing the network layer services required to connect to the network. Ishii and Francis give a similar justification in [17]. In their approach an observer is placed at the output of the plant to reconstruct

the state vector. The result is then quantized and sent over the network to the controller.

The observer has as inputs the output and input of the plant. In the implementation of this setup, to acquire the input of the plant, which is at the other side of the communication link, the observer can have a version of the model and controller and knowledge of the update time h. In this way, the output of the controller, that is, the input of the plant, can be simultaneously and continuously generated at both ends of the feedback path. The only requirement is that the observer must ensure that the model has been updated. This last requirement ensures that both the controller and the observer are synchronized. The handshaking provided by most network protocols can be used.

Again, we use the plant model, $\dot{\hat{x}} = \hat{A}\hat{x} + \hat{B}u$, to design the state observer. See Fig. 2.2. The observer has the form of a standard state observer with gain L. In summary, the system dynamic equations are, for $t \in [t_k, t_{k+1})$,

$$\begin{array}{ll} \text{Plant:} & \dot{x} = Ax + Bu,\ y = Cx + Du \\ \text{Model:} & \dot{\hat{x}} = \hat{A}\hat{x} + \hat{B}u, y = \hat{C}\hat{x} + \hat{D}u \\ \text{Controller:} & u = K\hat{x} \\ \text{Observer:} & \dot{\bar{x}} = (\hat{A} - L\hat{C})\bar{x} + \left[\hat{B} - L\hat{D}\ \ L\right]\left[u^T\ \ y^T\right]^T. \end{array}$$

We now proceed in a similar way as in the previous case of full feedback. There will be an update interval h, after which the observer updates the controller's model state $\hat{x}$ with its estimate $\bar{x}$. Define an error e that will be the difference between the controller's model state and the observer's estimate: $e = \bar{x} - \hat{x}$. Also define the modeling error matrices in the same way as before: $\tilde{A} = A - \hat{A}$, $\tilde{B} = B - \hat{B}$, $\tilde{C} = C - \hat{C}$, and $\tilde{D} = D - \hat{D}$. Define $z = \left[x^T\ \ \bar{x}^T\ \ e^T\right]^T$, and

$$\Lambda_o = \begin{bmatrix} A & BK & -BK \\ LC & \hat{A} - L\hat{C} + \hat{B}K + L\tilde{D}K & -\hat{B}K - L\tilde{D}K \\ LC & L\tilde{D}K - L\hat{C} & \hat{A} - L\tilde{D}K \end{bmatrix}.$$

Theorem 2 *[13] The output feedback MB-NCS is globally exponentially stable around the solution* $z = \left[x^T\ \ \bar{x}^T\ \ e^T\right]^T = 0$ *if and only if the eigenvalues of* $M_o = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & 0 \end{bmatrix} e^{\Lambda_o h} \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & 0 \end{bmatrix}$ *are inside the unit circle.*

The eigenvalues of the test matrix M_o can be studied in a similar fashion as in the state feedback case. By replacing h by t, applying the Laplace transform, and isolating the nonzero upper left block of M_o we obtain

$$P = \begin{bmatrix} \left(sI - \hat{A} - \hat{B}K\right)^{-1} + \Delta_1 & (sI - A)^{-1}BK\left(sI - \hat{A} - \hat{B}K\right)^{-1} \\ \Delta_3 & \left(sI - \hat{A} + L\hat{C}\right)^{-1} + \Delta_2 \end{bmatrix}$$

with

$$\Delta_1 = (sI - A)^{-1}(\tilde{A} + \tilde{B}K)(sI - \hat{A} - \hat{B}K)^{-1}$$

$$\Delta_2 = (sI - \hat{A} + L\hat{C})^{-1}\left(\tilde{B}K - \left(\tilde{A} - L\tilde{C}\right)(sI - A)^{-1}BK\right)(sI - \hat{A} - \hat{B}K)^{-1}$$

$$\Delta_3 = (sI - \hat{A} + L\hat{C})^{-1}\left(\tilde{A} - L\tilde{C} + \tilde{B}K - \left(\tilde{A} - L\tilde{C}\right)(sI - A)^{-1}\left(\tilde{A} + \tilde{B}K\right)\right) \cdot (sI - \hat{A} - \hat{B}K)^{-1}$$

It is clear that if $\tilde{A} \to 0$, $\tilde{B} \to 0$, and $\tilde{C} \to 0$ then $\Delta_1 \to 0$, $\Delta_2 \to 0$, and $\Delta_3 \to 0$. By doing so we obtain

$$P_L = \lim_{\Delta \to 0} P = \begin{bmatrix} \left(sI - \hat{A} - \hat{B}K\right)^{-1} & (sI - A)^{-1}BK\left(sI - \hat{A} - \hat{B}K\right)^{-1} \\ 0 & \left(sI - \hat{A} + L\hat{C}\right)^{-1} \end{bmatrix}.$$

From here it can be seen that by making the error between the plant and the model zero, the system's compensated eigenvalues would be placed at $\text{eig}\left\{e^{(\hat{A}+\hat{B}K)h}\right\} \cup \text{eig}\left\{e^{(\hat{A}-L\hat{C})h}\right\}$. Similarly to the full state feedback case, there are perturbation matrices Δ_1, Δ_2, and Δ_3 that can be made small by reducing the error between the actual plant and the model dynamic equations. The perturbation is over a matrix that has the eigenvalues of the compensated model. Finally, note that the separation principle of classic control, where controller and observer can be designed independently, cannot be used here.

2.3 State feedback MB-NCS with network-induced delays

Previously we assumed that the network delays were negligible. This is usually true for plants with slow dynamics relative to the network bandwidth. When this is not the case, the network delay cannot be neglected. Network delays can occur for many reasons. There are three important delay sources:

- Processing time
- Media access contention
- Propagation and transmission time.

The first one, processing time, occurs at both ends of the communication channel. On the transmitter, the processing time is the time elapsed between the time at which the transmitting process makes the request to the operating system to transmit a message, to the time at which the message is ready to be sent. On the receiver this is the time interval that occurs between the time at which the last bit of the message is received by the receiver, and the time at which the message is delivered by the operating system to the receiver process.

The media access contention time is the length of time the transmitter has to wait until the communication channel is not busy. This is usually the case when several transmitters have to share the same media.

The propagation and transmission time is the time the message takes to be placed on the network media and to travel through the network to reach the receiver. In local area networks the time the message takes to travel or propagate through the media is small in comparison to that for wide area networks or internetworks like the Internet. The time the message takes to be placed on the network depends on the size of the message and the baud rate.

If the control network is a local area network, as is common practice in industry, the propagation and transmission time can be established beforehand with good accuracy, and the same is true for the processing time. If real-time operating systems are used, the processing time can be accurately calculated. Finally, the media access contention delay can be fixed with the use of a communication protocol with scheduling. Fast data communication networks like Token Ring, Token Bus, and ArcNet fall into this category. Industry-oriented control networks like Foundation Fieldbus also implement a scheduler through their link active scheduler (LAS) . Even the inherently non-deterministic Ethernet has addressed the problem of not having a specified contention time with the so-called switched Ethernet.

In conclusion, most of these delays can at least be bounded if the network conditions are appropriate.

In the following, we extend our results to include the case where transmission delay is present. We will assume that the update time h is larger than the delay time τ. As before, we will assume that the update time h and delay τ are constant. We will present here the case of full state feedback systems.

So, at times $kh - \tau$ the sensor transmits the state data to the controller/actuator. This data will arrive τ seconds later. Therefore, at times kh the controller/actuator receives the state vector value $x\,(kh - \tau)$. The main idea is to use the plant model in the controller/actuator to calculate the present value of the state. After this, the state estimate obtained can be used to update the controller's model as in previous setups. The system is depicted in Fig. 6.

The propagation unit uses the plant model and the past values of the control input $u(t)$ to calculate an estimate of actual state $\breve{x}(kh)$ from the received data $x(kh - \tau)$. This estimate is then used to update the model which the controller will generate the control signal for the plant.

The system is described by the following equations:

$$\begin{array}{ll}
\text{Plant:} & \dot{x} = Ax + Bu \\
\text{Model:} & \dot{\hat{x}} = \hat{A}\hat{x} + \hat{B}u \\
\text{Controller:} & u = K\hat{x},\ t \in [t_k, t_{k+1}) \\
\text{Propagation unit:} & \dot{\breve{x}} = \hat{A}\breve{x} + \hat{B}u,\ t \in [t_{k+1} - \tau, t_{k+1}] \\
\text{Update law:} & \breve{x} \leftarrow x,\ t = t_{k+1} - \tau \text{ and } \hat{x} \leftarrow \breve{x},\ t = t_{k+1}.
\end{array} \tag{2}$$

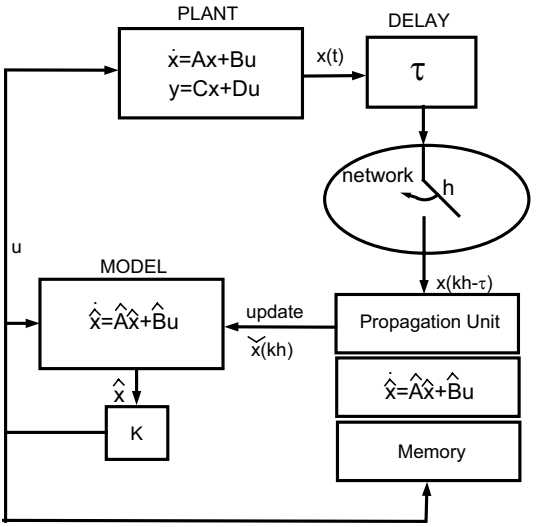


Fig. 6. Proposed configuration of a state feedback NCS in the presence of network delays

We define the errors $\hat{e} = \breve{x} - \hat{x}$ and $\breve{e} = x - \breve{x}$. We also make the following definitions:

$$\begin{matrix} \tilde{A} = A - \hat{A} \\ \tilde{B} = B - \hat{B} \end{matrix}, \quad \Lambda_d = \begin{bmatrix} A + BK & -BK & -BK \\ \tilde{A} + \tilde{B}K & \hat{A} - \tilde{B}K & -\tilde{B}K \\ 0 & 0 & \hat{A} \end{bmatrix}, \quad z = \begin{bmatrix} x \\ \breve{e} \\ \hat{e} \end{bmatrix}.$$

Theorem 3 *[13] The state feedback MB-NCS with networked-induced delay τ is globally exponentially stable around the solution $z = \begin{bmatrix} x^T & \breve{e}^T & \hat{e}^T \end{bmatrix}^T = 0$ if and only if the eigenvalues of $M_T = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & 0 \end{bmatrix} e^{\Lambda_d \tau} \begin{bmatrix} I & 0 & 0 \\ 0 & 0 & 0 \\ 0 & I & I \end{bmatrix} e^{\Lambda_T (h-\tau)}$ are inside the unit circle.*

It is interesting to note that the results of Theorem 3 can be seen as a generalization of Theorem 1. This can be shown by driving τ to zero.

Example 3 *Here we present a numeric example with the same plant that we have been using, $A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$, with randomly generated plant model $\hat{A} = \begin{bmatrix} -0.3444 & 0.9225 \\ -0.3089 & 0.3560 \end{bmatrix}$, $\hat{B} = \begin{bmatrix} -0.0098 \\ 1.3159 \end{bmatrix}$, and controller $K = \begin{bmatrix} -1 & -2 \end{bmatrix}$. Fig. 7 shows the plots for the maximum eigenvalue magnitude as a function of h for three different values of τ. The maximum value h can have to preserve stability is reduced when τ is increased.*

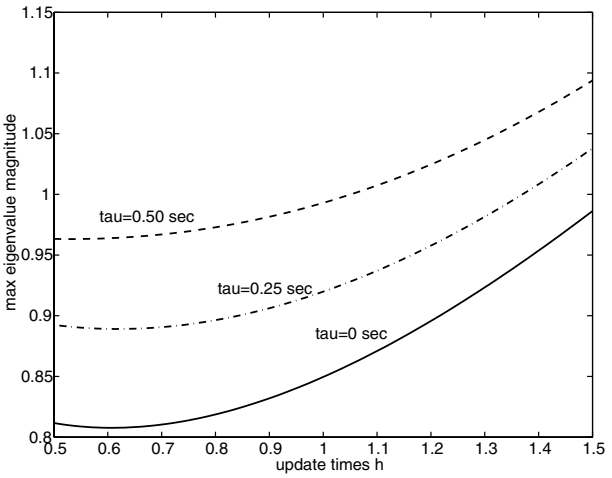


Fig. 7. Maximum eigenvalue of test matrix M versus the update times h for $\tau = 0$, 0.25, and 0.50 sec

3 A Performance Index for Linear MB-NCS with Constant Update Times

The performance characterization of NCSs under different conditions is also of major concern, along with stability. It is clear that, since the MB-NCS is h-periodic, there is no transfer function in the normal sense whose H2 norm can be calculated [2]. For linear time-invariant (LTI) systems the H2 norm can be computed by obtaining the 2-norm of the impulse response of the system. We will extend this definition to specify an H2 norm, or more accurately, to define an H2-like performance index [2]. We will call this performance index the *extended H2 norm*. We will study the extended H2 norm of the MB-NCS with output feedback studied in Section 2.2. A disturbance signal w and a performance or objective signal z are included in the setup.

We will start by defining the system dynamics.

$$\begin{array}{ll} \text{Plant dynamics:} & \text{Observer dynamics:} \\ \dot{x} = Ax + B_1 w + B_2 u & \dot{\bar{x}} = (\hat{A} - L\hat{C}_2)\bar{x} + (\hat{B}_2 - L\hat{D}_{22})u + Ly \\ z = C_1 x + D_{12} u & \text{Model dynamics:} \\ y = C_2 x + D_{21} w + D_{22} u & \dot{\hat{x}} = \hat{A}\hat{x} + \hat{B}_2 u \\ & \text{Controller:} \\ & u = K\hat{x} \end{array} \tag{3}$$

Define the following:

$$\Lambda = \begin{bmatrix} A & B_2 K & -B_2 K \\ LC_2 & \hat{A} - L\hat{C}_2 + \hat{B}_2 K + L\tilde{D}_{22} K & -\hat{B}_2 K - L\tilde{D}_{22} K \\ LC_2 & L\tilde{D}_{22} K - L\hat{C}_2 & \hat{A} - L\tilde{D}_{22} K \end{bmatrix}$$

$$M(h) = \begin{bmatrix} I & 0 & 0 \\ 0 & I & 0 \\ 0 & 0 & 0 \end{bmatrix} e^{\Lambda h},\ B_N = \begin{bmatrix} B_1 \\ LD_{21} \\ LD_{21} \end{bmatrix},\ C_N = \begin{bmatrix} C_1 & D_{12}K & -D_{12}K \end{bmatrix}.$$

Theorem 4 *[21] The extended H2 norm, $\|G\|_{xh2}$, of the output feedback MB-NCS described in (3) is given by $\|G\|_{xh2} = trace\left(B_N^T X B_N\right)$ where X is the solution of the discrete Lyapunov equation $M(h)^T X M(h) - X + W_o(0,h) = 0$ and $W_o(0,h)$ is the observability Gramian computed as $W_o(0,h) = \int_0^h e^{\Lambda^T t} C_N^T C_N e^{\Lambda t} dt$.*

Note that the observability Gramian can be factorized as ${C_{aux}}^T C_{aux} = \int_0^h e^{\Lambda^T t} C_N^T C_N e^{\Lambda t} dt$. This allows one to compute the extended norm as the regular H2 norm of a discrete LTI system.

Corollary 1 *[21] Define ${C_{aux}}^T C_{aux} = \int_0^h e^{\Lambda^T t} C_N^T C_N e^{\Lambda t} dt$ and the auxiliary discrete system G_{aux} with parameters $A_{aux} = M(h)$, $B_{aux} = B_N$, C_{aux} and $D_{aux} = 0$; then the following holds:*

$$\|G\|_{xh2} = \|G_{aux}\|_2 .$$

Example 4 *We now present an example using a double integrator as the plant. Let the plant dynamics be given by $A = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}$, $B_1 = \begin{bmatrix} 0.1 \\ 0.1 \end{bmatrix}$, $B_2 = \begin{bmatrix} 0 & 1 \end{bmatrix}^T$, $C_1 = \begin{bmatrix} 0.1 & 0.1 \end{bmatrix}$, $C_2 = \begin{bmatrix} 1 & 0 \end{bmatrix}$, $D_{11} = 0$, $D_{12} = 0.1$, $D_{21} = 0.1$, $D_{22} = 0$. We will use the state feedback controller $K = \begin{bmatrix} -1 & -2 \end{bmatrix}$. A state estimator with gain $L = \begin{bmatrix} 20 & 100 \end{bmatrix}^T$ is used to place the state observer eigenvalues at –10. We will use a plant model with the following parameters: $\hat{A} = \begin{bmatrix} 0.1634 & 0.8957 \\ -0.1072 & -0.1801 \end{bmatrix}$, $\hat{B}_2 = \begin{bmatrix} -0.1686 \\ 1.0563 \end{bmatrix}$, $\hat{C}_2 = \begin{bmatrix} 0.8764 & 0.1375 \end{bmatrix}$, $\hat{D}_{22} = -0.1304$.*

In Fig. 8 we plot the extended H2 norm of the system as a function of the update times. Note that as the update time of the MB-NCS approaches zero, the value of the extended H2 norm approaches the H2 norm of the non-networked compensated system. Also note the performance degradation as the update time h is increased.

4 Stability of MB-NCS with Time-Varying Update Times

In this section we relax our assumption that the update times $h(k)$ are constant. Here we study the state feedback MB-NCS shown in Fig. 1. The packets transmitted by the sensor contain the measured value of the plant state and are used to update the plant model on the actuator/controller node. These

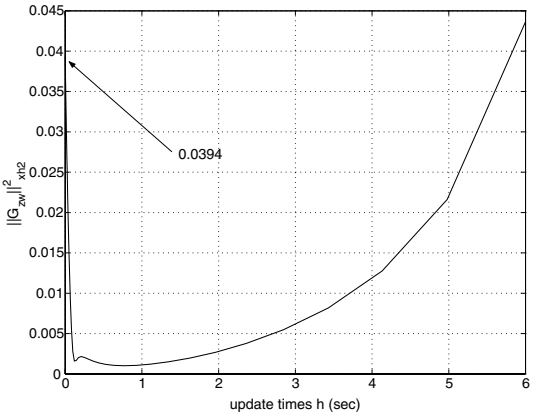


Fig. 8. Extended H2 norm of the system as a function of the update times

packets are transmitted at times t_k . We define the update times as the times between transmissions or model updates: $h(k) = t_{k+1} - t_k$. Previously, we made the assumption that the update times $h(k)$ are constant. This might not always be the case in applications. The transmission times of data packets from the plant output to the controller/actuator might not be completely periodic due to network contention and the usual non-deterministic nature of the transmitter task execution scheduler. Soft real-time constraints provide a way to enforce the execution of tasks in the transmitter microprocessor. This allows the task of periodically transmitting the plant information to the controller/actuator to be executed at times t_k that can vary according to certain probability distribution functions. This translates into an update time $h(k)$ that can acquire a certain value according to a probability distribution function. Most work on NCSs assumes deterministic communication rates [17,20] or time-varying rates without considering the stochastic behavior of these rates [9, 19]. Little work has concentrated on characterizing stability or performance on an NCS under time-varying, stochastic communication.

We first study the stability properties of the feedback MB-NCS assuming that the update times can take any values in an interval $[h_{\min}, h_{\max}]$. In this case we will assume that we don't have any statistical knowledge about the update times. We analyze the stability properties of this system using Lyapunov techniques. This is the strongest type of stability presented and will provide a first cut on the characterization of the stability properties, perhaps for comparison purposes.

Next, two types of stochastic stability are discussed, *almost sure* or *probability-1 asymptotic stability* and *mean square* or *quadratic asymptotic stability*. The first one is the one that most closely resembles deterministic stability [3]. Mean square stability is attractive since it is related to optimal control problems such as the linear quadratic regulator (LQR).

Two different types of time-varying transmission times are considered for each stochastic stability criterion. The first assumes that the transmission times are independent and identically distributed with a probability distribution function that may have support for infinite update times. The second type of stochastic update time assumes that the transmission times are driven by a finite Markov chain. Both models are common ways of representing the behavior of network transmission and scheduler execution times.

4.1 Lyapunov stability of MB-NCS

The stability criterion derived in this section is the strongest and most conservative stability criterion. It is based on the well-known Lyapunov second method for determining the stability of a system. We will assume that the properties of $h(k)$ are unknown but $h(k)$ is contained within some interval. This criterion is not stochastic but provides a first approach to stability for a time-varying transmission times NCS.

Definition 1 *The equilibrium* $z = 0$ *of a system described by* $\dot{z} = f(t, z)$ *with initial condition* $z(t_0) = z_0$ *is Lyapunov asymptotically stable at large (or globally) if for any* $\varepsilon > 0$ *there exists* $\beta > 0$ *such that the solution of* $\dot{z} = f(t, z)$ *satisfies* $\|z(t, z_0, t_0)\| < \varepsilon, \forall t > t_0$ *and* $\lim_{t \to \infty} \|z(t, z_0, t_0)\| = 0$ *whenever* $\|z_0\| < \beta$.

Theorem 5 *[14] The state feedback MB-NCS is Lyapunov asymptotically stable for* $h \in [h_{\min}, h_{\max}]$ *if there exists a symmetric positive definite matrix* X *such that* $Q = X - MXM^T$ *is positive definite for all* $h \in [h_{\min}, h_{\max}]$ *where* $M = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} e^{\Lambda h} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}$.

Theorem 5 may be used to derive an interval $[h_{\min}, h_{\max}]$ for h for which stability is guaranteed. It is clear that the range for h, that is, the interval $[h_{\min}, h_{\max}]$, will vary with the choice of X. Another observation is that the interval obtained this way will always be contained in the set of constant update times for which the system is stable (as derived using Theorem 1). That is, an update time contained in the interval $[h_{\min}, h_{\max}]$ will always be a stable constant update time.

Several ways of obtaining the values for $h_{\min}$ and $h_{\max}$ can be used. One is to first fix the value of Q, and obtain the solution X of the Lyapunov equation in Theorem 5 for a value of h known to be stable. Then, using this value of X, the expression $X - MXM^T$ can be evaluated for positive definiteness. This can be repeated for all the values of h known to stabilize the system to obtain the widest interval $[h_{\min}, h_{\max}]$.

4.2 Almost sure or probability-1 asymptotic stability

We will use the definition of almost sure asymptotic stability [3] that provides a stability criterion based on the sample path. This stability definition resembles more the deterministic stability definition [4], and it is of practical importance. Since the stability condition has been relaxed, we expect to see less conservative results than those obtained using the Lyapunov stability considered in the previous section. We now define *almost sure* or *probability-1 asymptotic stability.*

Definition 2 *The equilibrium* $z = 0$ *of a system described by* $\dot{z} = f(t, z)$ *with initial condition* $z(t_0) = z_0$ *is almost sure (or with probability-1) asymptotically stable at large (or globally) if for any* $\beta > 0$ *and* $\varepsilon > 0$ *the solution of* $\dot{z} = f(t, z)$ *satisfies* $\lim_{\delta \to \infty} P\left\{ \sup_{t \geq \delta} \|z(t, z_0, t_0)\| > \varepsilon \right\} = 0$ *whenever* $\|z_0\| < \beta$.

This definition is similar to the one presented for deterministic systems in Definition 1. We will examine the conditions under which the full state feedback continuous networked system in Fig. 1 is stable.

MB-NCS with independent and identically distributed transmission times

Here we will assume that the update times $h(k)$ are independent and identically distributed (iid) with probability distribution function $F(h)$. We now present the conditions under which the full state feedback MB-NCS with iid update times is asymptotically stable with probability-1. We will use a technique similar to lifting [2] to obtain a discrete LTI representation of the system. It can be observed that the system can be described by

$$\xi_{k+1} = \Omega_k \xi_k, \text{ with } \xi_k \in L_{2e} \text{ and } \xi_k = z(t + t_k),\ t \in [0, h_k). \tag{4}$$

Here L_{2e} stands for the extended L_2. It can be shown that the operator Ω_k can be represented as

$$(\Omega_k \nu)(t) = e^{\Lambda t} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} \int_0^{h(k)} \delta(\tau - h(k)) \nu(\tau) d\tau, \tag{5}$$

where $\delta(t)$ represents the impulse function. Now we can restate the definition on almost sure stability or probability-1 stability given in Definition 2 to better fit the equivalent system representation (4).

Definition 3 *The system represented by (4) is almost sure stable or stable with probability-1 if for any* $\beta > 0$ *and* $\varepsilon > 0$ *the solution of* $\xi_{k+1} = \Omega_k \xi_k$

satisfies $\lim_{\tilde{\delta}\to\infty} P\left\{\sup_{k\geq\tilde{\delta}} \|\xi_k(t_0,z_0)\|_{2,[0,t_k]} > \varepsilon\right\} = 0$ *whenever* $\|z_0\| < \beta$. *Here the norm* $\|\cdot\|_{2,[0,h(k)]}$ *is given by* $\|\xi_k\|_{2,[0,h(k)]} = \left(\int_0^{h(k)} \|\xi_k(\tau)\|^2 d\tau\right)^{1/2}$.

This definition allows us to study almost sure stability of systems such as (4) when the probability distribution function for update times has infinite support. Based on this definition, the following result can now be shown.

Theorem 6 *[14] The state feedback MB-NCS, with update times* $h(j)$ *that are independent identically distributed random variable with probability distribution* $F(h)$ *is globally almost sure (or with probability-1) asymptotically stable around the solution* $z = \begin{bmatrix} x^T & e^T \end{bmatrix}^T = 0$ *if* $N = E\left[\left(e^{2\bar{\sigma}(\Lambda)h} - 1\right)^{1/2}\right] < \infty$ *and the expected value of the maximum singular value of the test matrix* M, $E[\|M\|] = E[\bar{\sigma}_M]$, *is strictly less than one, where* $M = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} e^{\Lambda h} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}$.

Note that the condition may give conservative results if applied directly over the test matrix. To avoid this problem and make the condition tighter we may apply a similarity transformation over the test matrix M.

The condition on the matrix N ensures that the probability distribution function for the update times $F(h)$ assigns smaller occurrence probabilities to increasingly long update times, that is, $F(h)$ decays rapidly. In particular, we observe that N can always be bounded if there exists h_m such that $F(h) = 0$ for h larger than h_m. We can also bound the expression inside the expectation to obtain $E\left[\left(e^{2\bar{\sigma}(\Lambda)h} - 1\right)^{1/2}\right] < E\left[e^{\bar{\sigma}(\Lambda)h}\right]$ and formulate the following corollary.

Corollary 2 *The state feedback MB-NCS, with update times* $h(j)$ *that are independent identically distributed random variable with probability distribution* $F(h)$ *is globally almost sure (or with probability-1) asymptotically stable around the solution* $z = \begin{bmatrix} x^T & e^T \end{bmatrix}^T = 0$ *if* $T = E\left[e^{\bar{\sigma}(\Lambda)h}\right] < \infty$ *and the expected value of the maximum singular value of the test matrix* M, $E[\|M\|] = E[\bar{\sigma}_M]$, *is strictly less than one.*

Note that the Corollary 2 condition $T = E\left[e^{\bar{\sigma}(\Lambda)h}\right] < \infty$ is automatically satisfied if the probability distribution function $F(h)$ does not have infinite support. It otherwise indicates that $F(h)$ should roll off fast enough to counteract the growth of M's maximum singular value as h increases.

Example 5 *We use the unstable double integrator plant. We now assume that* $h(k)$ *is a random variable with a uniform probability distribution function* $U(0.5,\ h_{\max})$. *The plot of the expected maximum singular value of a similarity*

transformation of the original test matrix M is shown in Fig. 9. The similarity transformation used here was one that diagonalizes the matrix M for $h = 1$.

We see that the maximum value for $h_{\max}$ is around 1.3 seconds (maximum constant update time for stability is $h = 1$ second). So we see that the double integrator with uniformly distributed update time between 0.5 and 1.3 seconds is stable, while the same system but with a constant update time of 1 second is unstable. This also represents an improvement over the result that we may have obtained by using the previously discussed Lyapunov stability condition in which the maximum update time obtainable would have been less than 1 second.

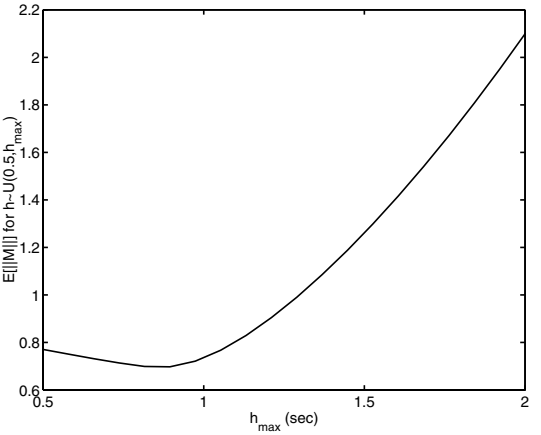


Fig. 9. Average maximum singular value for $h \sim U(0.5, h_{\max})$ as a function of $h_{\max}$, zero dynamics plant model

The advantage of using a model-based approach resides in its ability to reduce the amount of bandwidth required. The previous example shows stability conditions for a model that represents a zero-order hold, that is, the control value is kept constant until the next update time. We will now show the same plots for a model that better resembles the plant; this was done by randomly perturbing the plant matrices. The plant model matrices are $\hat{A} = \begin{bmatrix} 0.0844 & 0.9353 \\ 0.0476 & -0.0189 \end{bmatrix}$, $\hat{B} = \begin{bmatrix} 0.0871 \\ 1.0834 \end{bmatrix}$. *Fig. 10 shows that stability is maintained for update times that have uniform distribution with a max update time of 5.5 seconds. This shows that improved knowledge over the plant dynamics can translate into a significant improvement in terms of stability.*

MB-NCS with Markov chain-driven transmission times

In certain cases it is appropriate to represent the dynamics of the update times as driven by a Markov chain. A good example of this is when the network

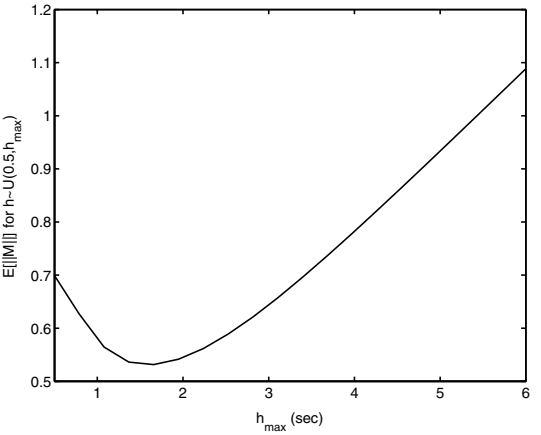


Fig. 10. Average maximum singular value for $h \sim U\,(0.5,\, h_{\max})$ as a function of $h_{\max}$, improved plant model

experiences traffic congestion or has queues for message forwarding. We now present a stability criterion for the model-based control system in which the update times $h(k)$ are driven by a finite state Markov chain. Assume that the update times can take a value from a finite set:

$$h\,(k) \in \{h_1,\, h_2,\, ...,\, h_N\} \text{ and } h_i \neq \infty,\, \forall i \in [1, N]. \tag{6}$$

Let us represent the Markov chain process by $\{\omega_k\}$with state space $\{1, 2, ..., N\}$ and transition probability matrix Γ and $N \times N$ matrix with elements $p_{i,j}$ and initial state probability distribution $\Pi_0 = \left[\pi_1\ \pi_2\ ...\ \pi_N\right]^T$. The transition probability matrix entries are defined as $p_{i,j} = \mathrm{P}\{\omega_{k+1} = j | \omega_k = i\}$. We can now represent the update times more appropriately as $h(k) = h_{\omega_k}$.

A sufficient condition for the almost sure stability of the system under Markovian jumps is given in the following theorem.

Theorem 7 *[14] The state feedback MB-NCS with update times* $h(k) = h_{\omega_k} \neq \infty$ *driven by a finite state Markov chain* $\{\omega_k\}$ *with state space* $\{1, 2, ..., N\}$ *and transition probability matrix* Γ *with elements* $p_{i,j}$ *and initial state probability distribution* $\Pi_0 = \left[\pi_1\ \pi_2\ ...\ \pi_N\right]^T$ *is globally almost sure asymptotically stable around the solution* $z = \left[x^T\ e^T\right]^T = 0$ *if the matrix* T *has all its eigenvalues inside of the unit circle, where*

$$T = \begin{bmatrix} \|M|_{h_1}\| & 0 & 0 & 0 \\ 0 & \|M|_{h_2}\| & 0 & 0 \\ 0 & 0 & ... & 0 \\ 0 & 0 & 0 & \|M|_{h_N}\| \end{bmatrix} \Gamma^T \ \ ;\ \ M|_{h_i} = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} e^{\Lambda h_i} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}.$$

If Γ is irreducible it follows that, since $\|M\|$ is non-negative, T is also irreducible. Then it can be shown using the Perron–Frobenius theorem as in [6], that T's maximum magnitude eigenvalue is real and is sometimes referred to as the Perron–Frobenius eigenvalue.

4.3 Mean square or quadratic asymptotic stability

We now define the type of stability called mean square asymptotic stability.

Definition 4 *The equilibrium $z = 0$ of a system described by $\dot{z} = f(t, z)$ with initial condition $z(t_0) = z_0$ is mean square stable asymptotically stable at large (or globally) if the solution of $\dot{z} = f(t, z)$ satisfies $\lim_{t\to\infty} E\left[\|z(t, z_0, t_0)\|^2\right] = 0$.*

A system that is mean square stable will have the expectation of system states converging to zero with time in the mean square sense. This definition of stability is attractive since many optimal control problems use the squared norm in their formulations. We will study the two cases of the previous section under this new stability criterion.

MB-NCS with independent and identically distributed transmission times

We present the conditions under which the state feedback networked control system is mean square stable, and we also discuss how these conditions relate to the ones for probability-1 stability.

Theorem 8 *[14] The state feedback MB-NCS with update times $h(j)$ that are independent identically distributed random variable with probability distribution $F(h)$ is globally mean square asymptotically stable around the solution $z = \begin{bmatrix} x^T & e^T \end{bmatrix}^T = 0$ if $K = E\left[\left(e^{\bar{\sigma}(\Lambda)h}\right)^2\right] < \infty$ and the maximum singular value of the expected value of $M^T M$, $\left\|E\left[M^T M\right]\right\| = \bar{\sigma}\left(E\left[M^T M\right]\right)$, is strictly less than one, where $M = \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix} e^{\Lambda h} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}$.*

We note the similarity between the conditions given by Theorems 6 and 8. For the first one we require the expectation of the maximum singular value of the test matrix to be less than one. However, for the second stability criterion it is required to have the maximum singular value of the expectation of the test matrix (multiplied by its transpose) to be less than one.

MB-NCS with Markov chain-driven transmission times

The type of stability criteria above depends on our ability to find appropriate $P(i)$ matrices. Several other results in jump system stability [5, 7] can be

extended to obtain other conditions on stability of NCSs. Note though, that most of the results available in the literature deal with similar but not identical types of systems.

Theorem 9 *The state feedback MB-NCS with update times* $h(k) = h_{\omega_k} \neq \infty$ *driven by a finite state Markov chain* $\{\omega_k\}$ *with state space* $\{1, 2, ..., N\}$ *and transition probability matrix* Γ *with elements* $p_{i,j}$ *is globally mean square asymptotically stable around the solution* $z = \begin{bmatrix} x^T & e^T \end{bmatrix}^T = 0$ *if there exist positive definite matrices* $P(1)$, $P(2)$, $\ldots$, $P(N)$ *such that*

$$\left(\sum_{j=1}^{N} p_{i,j} \left(H(i)^T P(j) H(i) \right) - P(i) \right) < 0, \forall i, j = 1...N$$

with $H(i) = e^{\Lambda h_i} \begin{bmatrix} I & 0 \\ 0 & 0 \end{bmatrix}$.

5 Stability of Linear MB-NCS with Quantization

Here we extend our stability results to consider the case where quantization errors occur. In particular, a state space MB-NCS is considered. Two static quantizers are studied: the *uniform quantizer* and the *logarithmic quantizer*. These are called static since they partition the state space into invariant regions that are fixed in time. We also note that these quantizers represent two of the most common data representations: fixed-point format for uniform quantizers and floating-point format for logarithmic quantizers. We will assume that the transmission times are constant. We will also assume that the compensated networked system *without* quantization is stable, thus in view of Theorem 1 there exists positive definite P that satisfies

$$\left(e^{\left(\hat{A}+\hat{B}K\right)^T h} + \Delta(h)^T \right) P \left(e^{\left(\hat{A}+\hat{B}K\right) h} + \Delta(h) \right) - P = -Q_D, \tag{7}$$

where $\Delta(h) = e^{Ah} \int_0^h e^{-A\tau} (\tilde{A} + \tilde{B}K) e^{(\hat{A}+\hat{B}K)\tau} d\tau$ and with Q_D symmetric and positive symmetric. Note that $e^{\left(\hat{A}+\hat{B}K\right)h} + \Delta(h)$ was previously defined in (1).

5.1 State feedback MB-NCS with uniform quantization

Define the uniform quantizer as a function $q : \mathrm{R}^n \rightarrow \mathrm{R}^n$ with the following property, $\|z - q(z)\| \leq \delta,\ z \in \mathrm{R}^n, \delta > 0$.

Theorem 10 *[21] The plant state of the state feedback MB-NCS satisfying (7) and using the uniform quantizer will enter and remain in the region* $\|x\| \leq R$ *defined by*

$$R = \left(e^{\bar{\sigma}\left(\hat{A}+\hat{B}K\right)h} + \Delta_{\max}(h)\right) r + \left(e^{\bar{\sigma}(A)h} + \Delta_{\max}(h)\right)\delta$$
$$\text{where } r = \sqrt{\frac{\lambda_{\max}\left(\left(e^{Ah}-\Delta(h)\right)^T P\left(e^{Ah}-\Delta(h)\right)^T\right)\delta^2}{\lambda_{\min}(Q_D)}}$$
$$\text{and } \Delta_{\max}(h) = \int_0^h e^{\bar{\sigma}(A)(h-\tau)}\bar{\sigma}\left(\tilde{A}+\tilde{B}K\right)e^{\bar{\sigma}\left(\hat{A}+\hat{B}K\right)\tau}d\tau.$$

5.2 State feedback MB-NCS with logarithmic quantization

Define the logarithmic quantizer as a function $q: \mathrm{R}^n \to \mathrm{R}^n$ with the following property, $\|z - q(z)\| \leq \delta \|z\|,\ z \in \mathrm{R}^n, \delta > 0$.

Theorem 11 *[21] The state feedback MB-NCS satisfying (7) and using the logarithmic quantizer is exponentially stable if*

$$\delta < \sqrt{\frac{\lambda_{\min}(Q_D)}{\lambda_{\max}\left(\left(e^{Ah}-\Delta(h)\right)^T P\left(e^{Ah}-\Delta(h)\right)\right)}}.$$

Example 6 *For our example we will use the following plant model parameters:* $A = \begin{bmatrix} 0 & 1 \\ 1 & 3 \end{bmatrix}$, $B = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$. *The plant will be a perturbed version of our plant model:* $\hat{A} = \begin{bmatrix} -0.0689 & 0.9757 \\ 1.0396 & 3.0720 \end{bmatrix}$, $\hat{B} = \begin{bmatrix} 0.0707 \\ 1.0187 \end{bmatrix}$. *Both are unstable plants. The controller is designed using the plant model:* $u = \begin{bmatrix} -2 & -5 \end{bmatrix}\hat{x}$. *This controller places both eigenvalues of the compensated plant model at –1. Using an update time of* $h = 0.6$ *second, we will test two logarithmic quantizer functions,* $q1$ *with a mantissa word length of 12 bits and* $q2$ *with mantissa word length of 13 bits. Their relative errors for the two-dimensional space they will work on are for* $q1 : 0.33$ *and for* $q2 : 0.20$. *Initializing the plant at* $\begin{bmatrix} 2^T & 3^T \end{bmatrix}^T$, *we observe from Fig. 11 that the system working with quantizer* $q1\,(\delta = 0.33)$ *is unstable, while with* $q2\,(\delta = 0.20)$, *it is stable (Fig. 11). By using Theorem 11 and a* $Q_D = I$ *we obtain a maximum relative error of 0.1241.*

6 Stability of a Class of Non-Linear MB-NCS

We will determine sufficient conditions for the stability of a state feedback MB-NCS when the plant and controller are non-linear. Let the plant, plant model, and controller be given by

$$\text{plant } \dot{x} = f(x) + g(u); \quad \text{model } \dot{\hat{x}} = \hat{f}(x) + \hat{g}(u); \quad \text{controller } u = \hat{h}(\hat{x}). \quad (8)$$

Also, define $e = x - \hat{x}$ as the error between the plant state and the plant model state. From (8) we obtain

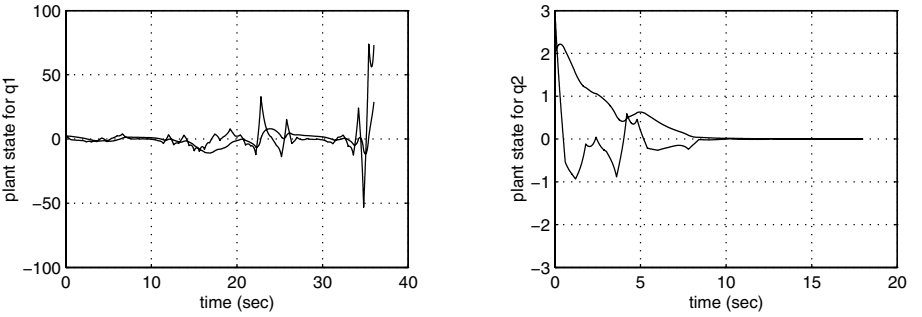


Fig. 11. Plant state time response for $q1$ and $q2$

$$\begin{aligned} \dot{x} &= f(x) + g\left(\hat{h}(\hat{x})\right) = f(x) + m(\hat{x}); \\ \dot{\hat{x}} &= \hat{f}(\hat{x}) + \hat{g}\left(\hat{h}(\hat{x})\right) = \hat{f}(\hat{x}) + \hat{m}(\hat{x}). \end{aligned} \tag{9}$$

We will also assume that the plant model dynamics differ from the actual plant dynamics in an additive fashion:

$$\hat{f}(\varsigma) = f(\varsigma) + \delta_f(\varsigma); \qquad \hat{m}(\varsigma) = m(\varsigma) + \delta_m(\varsigma). \tag{10}$$

So we can rewrite (9) as

$$\begin{aligned} \dot{x} &= f(x) + m(\hat{x}) \\ \dot{\hat{x}} &= f(\hat{x}) + m(\hat{x}) + \delta_f(\hat{x}) + \delta_m(\hat{x}) = f(\hat{x}) + m(\hat{x}) + \delta(\hat{x}). \end{aligned} \tag{11}$$

We will now assume that f and δ satisfy the following local Lipschitz conditions for $x, y \in B_L$ with B_L a ball centered on the origin:

$$\|f(x) - f(y)\| \leq K_f \|x - y\|; \qquad \|\delta(x) - \delta(y)\| \leq K_\delta \|x - y\|. \tag{12}$$

At this point it is to be noted that if the plant model is accurate, the Lipschitz constant K_δ will be small.

We will assume that the non-networked compensated plant model is exponentially stable when $\hat{x}(t_0) \in B_S$, with $\hat{x}(t) \in B_h$ for $t \in [t_0, t_0 + h)$ with B_S and B_h balls centered on the origin:

$$\|\hat{x}(t)\| \leq \alpha \|\hat{x}(t_0)\| e^{-\beta(t-t_0)}, \text{ with } \alpha, \beta > 0. \tag{13}$$

Theorem 12 *[21] The non-linear MB-NCS with dynamics described by (8), that satisfies the Lipschitz conditions described by (12) and with exponentially stable compensated plant model satisfying (13) is asymptotically stable if*

$$\left(1 - \alpha\left(e^{-\beta h} + \frac{K_\delta}{\beta}\left(e^{K_f h} - e^{-\beta h}\right)\left(\frac{\beta}{K_f + \beta}\right)\right)\right) > 0.$$

Theorem 12 presents a sufficient condition for stability of a class of non-linear systems. Note that if the model has the exact same dynamics as the plant, that is, if $K_\delta = 0$, then the condition will be satisfied for arbitrarily large h.

Example 7 *We use the inverted pendulum in Fig. 12 as an example.*

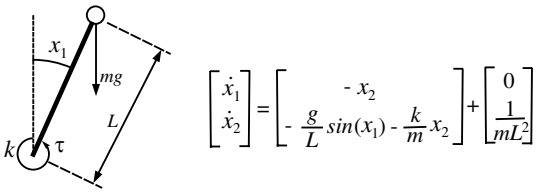


Fig. 12. Inverted pendulum

The parameters for the plant are $g = 10$, $L = 10$, $k = 0.1$, *and* $m = 1.01$. *The model parameters are the same as the plant ones, except for the mass, which is* $m = 1.00$. *Finally, the controller is given by* $\tau = \begin{bmatrix} -316 & 316 \end{bmatrix} \hat{x}$. *Using the following Lipschitz and exponential stability constants:* $K_f = 1.0507$, $K_\delta = 0.0450$, $\alpha = 1.5$, *and* $\beta = 0.6$, *Theorem 12 predicts stability for update times between 0.55 and 2.55 seconds. Simulations show that the system is unstable for* h *greater than approximately 4.5 seconds.*

7 Conclusions

The MB-NCS control architecture presented in this chapter represents a natural way of placing critical information about the plant on the network to reduce the data traffic load. By making the sensor and actuator more "intelligent" the NCS is able to predict the future behavior of the plant and send the precise information at critical times to ensure plant stability.

The MB-NCS is only one of the various approaches to networked control. We study these systems since their benefits and properties make them well suited for a variety of practical applications. Furthermore, the control structures presented appear to be amenable to detailed analysis.

Acknowledgements

The partial support of the Army Research Office (DAAG19-01-1-0743) and the National Science Foundation (NSF **CCR-02-08537** and **ECS-02-25265**) is gratefully acknowledged.

References

1. P. Antsaklis and A. Michel, *Linear Systems*, McGraw-Hill, New York, 1997.
2. T. Chen and B. Francis, *Optimal Sampled-Data Control Systems*, 2nd edition, Springer, London, 1996.
3. F. Kozin, A survey of stability of stochastic systems, *Automatica*, Vol. 5, pp. 95–112, 1968.
4. M. Mariton, *Jump Linear Systems in Automatic Control*, Marcel Dekker, New York, 1990.
5. O. L. V. Costa and M. D. Fragoso, Stability results for discrete-time linear systems with Markovian jumping parameters, *Journal of Mathematical Analysis and Applications*, Vol. 179, pp. 154–178, 1993.
6. A. Dembo and O. Zeitouni, *Large Deviation Techniques and Applications*, Springer-Verlag, New York, 1998.
7. Y. Fang, A new general sufficient condition for almost sure stability of jump linear systems, *IEEE Transactions on Automatic Control*, Vol. 42, No. 3, pp. 378–382, March 1997.
8. S. Woods, *Probability, Random Processes, and Estimation Theory for Engineers*, 2nd edition, Prentice-Hall, Upper Saddle River, NJ, 1994.
9. A. S. Matveev and A. V. Savkin, Optimal control of networked systems via asynchronous communication channels with irregular delays, 40th IEEE Conference on Decision and Control, Orlando Florida, December 2001, pp. 2323–2332.
10. L. A. Montestruque and P. J. Antsaklis, Model-based networked control systems: Stability, *ISIS Technical Report ISIS-2002-001*, University of Notre Dame, www.nd.edu/∼isis/, January 2002.
11. L. A. Montestruque and P. J. Antsaklis, Model-based networked control systems: necessary and sufficient conditions for stability, *10th Mediterranean Conference on Control and Automation*, July 2002.
12. L. A. Montestruque and P. J. Antsaklis, State and output feedback control in model-based networked control systems, *41st IEEE Conference on Decision and Control*, December 2002.
13. L. A. Montestruque and P. J. Antsaklis, On the model-based control of networked systems, *Automatica*, Vol. 39, pp. 1837–1843.
14. L. A. Montestruque and P. J. Antsaklis, Stability of networked systems with time varying transmission times, *IEEE Transactions of Automatic Control*, 2004, to appear.
15. G. N. Nair and R. J. Evans, Communication–limited stabilization of linear systems, *Proceedings of the IEEE Conference on Decision and Control,* 2000, pp. 1005–1010.
16. G. N. Nair and R. J. Evans, Networked control sessions, *Proceedings of the IEEE Conference on Decision and Control, Proceedings of the American Control Conference,* 2003.
17. H. Ishii and B. Francis, Stabilizing a linear system by switching control and output feedback with dwell time, *Proceedings of the 40th IEEE Conference on Decision and Control*, December 2001, pp. 191–196.
18. J. K. Yook, D. M. Tilbury, and N. R. Soparkar, Trading computation for bandwidth: Reducing communication in distributed control systems using state estimators, *IEEE Transactions on Control Systems Technology*, July 2002, Vol. 10, No. 4, pp. 503–518.

19. G. Walsh, H. Ye, and L. Bushnell, Stability analysis of networked control systems, *Proceedings of American Control Conference*, June 1999.
20. D. Hristu-Varsakelis, Feedback control systems as users of a shared network: Communication sequences that guarantee stability, *Proceedings of the 40th IEEE Conference on Decision and Control*, December 2001, pp. 3631–3636.
21. L. A. Montestruque and P. J. Antsaklis, Technical Reports, University of Notre Dame, www.nd.edu/∼isis/.

Control Issues in Systems with Loop Delays

Leonid Mirkin and Zalman J. Palmor

Faculty of Mechanical Engineering
Technion—IIT
Haifa 32000, Israel
{mirkin,palmor}@technion.ac.il

1 Introduction

This chapter discusses properties of feedback control systems containing loop delays (dead-time systems), and approaches to controller design for such systems. Consider the feedback system depicted in Fig. 1, where $\mathcal{P}$ is a plant, $\mathcal{C}$ is a controller, r is a reference signal, d is a disturbance, u is a control signal, and y is an output (measurement) signal. It is assumed throughout that both the measured signal y and the control signal u are delayed by h_y and h_u units of time, respectively. This is reflected in Fig. 1 by the two delay blocks containing the *delay element* $\mathcal{D}_h$ defined by

$$x_{\mathrm{o}}(t) = \mathcal{D}_h\, x_{\mathrm{i}}(t) \iff x_{\mathrm{o}}(t) = x_{\mathrm{i}}(t - h(t)).$$

It is readily seen that $\mathcal{D}_h$ is *linear* (superposition property holds) and, whenever h is constant, is *time invariant* and *BIBO stable* (a constant time shift changes neither the magnitude nor the energy of a signal). In the time-invariant case $\mathcal{D}_h$ can be described by transfer function formalism. It can be shown that the transfer function of $\mathcal{D}_h$ is e^{-sh} (in continuous time) or z^{-h} (in discrete time). Note that whereas the latter is a finite-dimensional (h-dimensional, to be precise) transfer function, the former is not as it is an irrational function of s.

Loop delays arise naturally in numerous control applications, both from delays in processes and control interfaces and from the use of delays to model

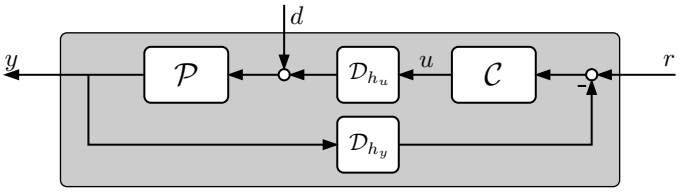


Fig. 1. Unity feedback system with loop delays (dead-time system)

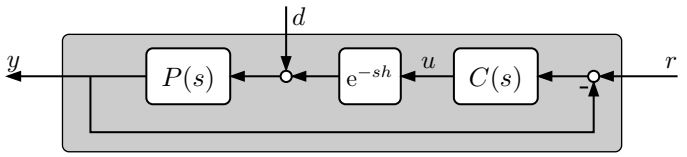


Fig. 2. Simplified unity feedback system with input delay

complicated physical phenomena and model reduction. In particular, loop delays are practically inevitable in systems controlled via communication networks as data is delayed due to buffering and propagation delays.

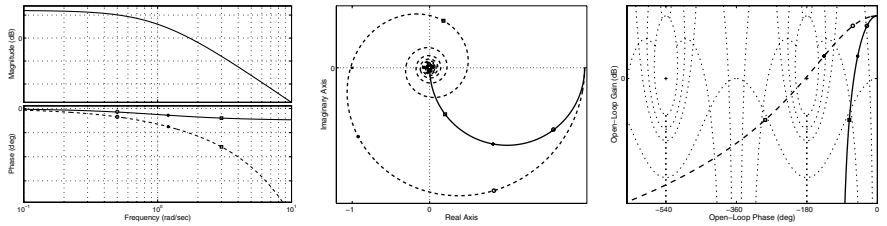
The presence of loop delays typically imposes strict limitations on achievable feedback performance. With delays in measurement channels, the controller receives "outdated" information about process behavior. Similarly, with delays in actuation channels, control action cannot be applied "on time" thus reducing the efficiency of the compensation of the effect of disturbances, etc.

The presence of loop delays also *complicates controller design* considerably. In the continuous-time case, complications are caused by the fact that the delay element is infinite dimensional, so that many classical methods cannot be applied directly. In the discrete-time case, the presence of the delay element is considered somewhat less technically challenging as the finite-dimensional z^{-h} can be easily absorbed into the plant. Yet this approach might be misleading as it typically increases the problem dimension and blurs the structure of the delay element.

The purpose of this chapter is to give a short exposition of problems arising in feedback control systems due to loop delays. To this end, the basic properties of dead-time systems will be described and some approaches to controller design for such systems will be presented. As our purpose here is to *provide a flavor of the underlying ideas*, we mostly limit the discussion to the simplest case of time-invariant single-input/single-output (SISO) systems in continuous time and attempt to avoid heavy mathematical details. For the same reason we consider only the input delay case, i.e., we assume that $h_y = 0$, see Fig. 2. For more detailed and general discussions the reader is refereed to [1–4] and the references therein.

2 Effects of Loop Delays on Closed-Loop Dynamics

In this section we study how the presence of loop delays affects the dynamics of closed-loop control systems. We will consider frequency-domain properties in connection with the Nyquist stability criterion and classical loop-shaping arguments (§2.1), location of the roots of the characteristic polynomial of the system in Fig. 2 (§2.2), the effect of the delay on the state-space realization of the system (§2.3), and some approaches to the rational approximation of the delay element (§2.4).



(a) Bode diagram (b) Nyquist diagram (c) Nichols chart

Fig. 3. Frequency-response plots of $\frac{2}{s+1}$ (solid lines) and $\frac{2}{s+1}\mathrm{e}^{-1.25s}$ (dashed lines)

2.1 Frequency response and Nyquist arguments

Arguably, the most "painless" incorporation of loop delays into the classical analysis takes place when frequency-domain Nyquist criterion arguments are applied to the system in Fig. 2. The reason is that these arguments are based on the *open-loop* frequency response, the effect of the loop delay on which is rather simple. Indeed, consider the loop transfer function of the system in Fig. 2:

$$L(s) = L_{\mathrm{r}}(s)\mathrm{e}^{-sh},$$

where the transfer function $L_{\mathrm{r}}(s) = P(s)C(s)$ is assumed to be rational (finite dimensional). It is readily seen that the frequency response of L is

$$L(j\omega) = L_{\mathrm{r}}(j\omega)\mathrm{e}^{-j\omega h} = |L_{\mathrm{r}}(j\omega)|\mathrm{e}^{j\arg L_{\mathrm{r}}(j\omega)}\mathrm{e}^{-j\omega h} = |L_{\mathrm{r}}(j\omega)|\mathrm{e}^{j(\arg L_{\mathrm{r}}(j\omega)-\omega h)}.$$

The magnitude of $L(j\omega)$ is *not affected* by the delay element. The latter, however, introduces an *additional phase lag* in $L(j\omega)$ that grows linearly with the frequency ω and is proportional to the delay h, namely, ωh radians.

The arguments above imply that the frequency response plots of $L(s)$ can be produced from those of its rational part $L_{\mathrm{r}}(s)$ using the following simple rules, which are illustrated by the plots in Fig. 3 for $L_{\mathrm{r}}(s) = \frac{2}{s+1}$ and $h = 0.25$.

Bode diagram: the magnitude plot does not change while the phase plot is shifted down by $\frac{180}{\pi}\,\omega h$ degrees (exponentially decaying function in the logarithmic scale).

Nyquist diagram: every point on the diagram is rotated clockwise by ωh radians.

Nichols chart: every point on the chart is shifted to the left by $\frac{180}{\pi}\,\omega h$ degrees.

The remarkable aspect of the stability analysis of dead-time systems using the frequency-response methods is that the Nyquist criterion is *literally applicable* to such systems. This fact was first noticed by Ya. Z. Tsypkin in 1946 and its proof follows by the principle of the argument much in parallel to the rational case.[1] For example, the Nyquist plot of the (stable) loop transfer

[1] Though certain care should be taken in the case when L_{r} is not strictly proper, see, e.g., [5].

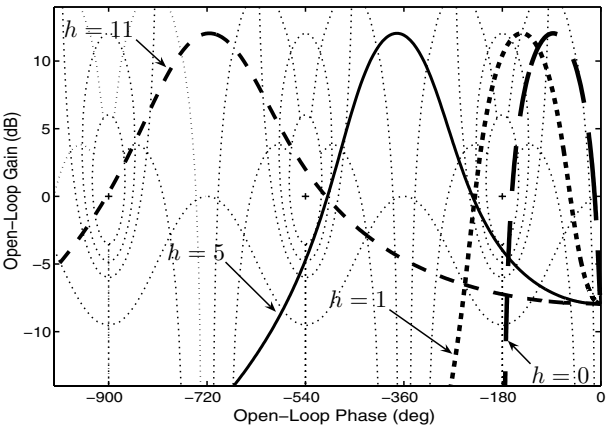


Fig. 4. Nichols charts of $\frac{0.4}{s^2+0.1s+1}\,\mathrm{e}^{-sh}$

function $\frac{2}{s+1}\,\mathrm{e}^{-1.25s}$ in Fig. 3(b) does encircle the critical point $(-1, 0)$, so that the corresponding closed-loop system is unstable.

Note that in the example above the delay has a *destabilizing effect* on the closed-loop dynamics. It is readily seen that the closed-loop system is stable for all delays smaller than some critical value, say h_σ (which is slightly smaller than 1.25 in the example), but unstable for all $h \geq h_\sigma$. Moreover, the larger h is, the smaller the stability margins are. Although this situation is encountered in many practical applications, it is not generic. In some systems stability and instability regions may interlace each other as illustrated by the following example.

Example 1. Consider the system in Fig. 2 with $L_{\mathrm{r}}(s) = \frac{0.4}{s^2+0.1s+1}$. The Nichols charts of this loop for different delays (taken from the set $\{0, 1, 5, 11\}$) are depicted in Fig. 4. It is readily seen that the closed-loop system is stable for $h = 0$ and $h = 5$ (solid lines) yet unstable for $h = 1$ and $h = 11$ (dashed lines). This behavior is a consequence of the fact that when the phase lag in the first (the smallest) crossover frequency ω_{c1} exceeds $-\pi$, the phase lag in the second (the largest) crossover frequency ω_{c2} is still smaller than -3π. In other words, when $h = 5$, the angular distance ψ_h between two crossover points is smaller than 2π, so that when the resonant peak is neatly placed in between two critical points, the Nichols plot encircles neither of the critical points.

Note that as the angular distance between two crossover points is a strictly increasing affine function of h ($\psi_h = \psi_0 + (\omega_{c2} - \omega_{c1})h$), there must exist a delay, say $\hat{h}$, such that $\psi_h \geq 2\pi$ for all $h \geq \hat{h}$. This implies that the interlacing of "stabilizing" and "destabilizing" delays always ends at some finite $h < \hat{h}$ and above that value delays are always destabilizing.

Example 1 shows that delay may have a stabilizing effect on the closed-loop dynamics. Some authors even suggest exploiting this property by artificially

adding delay elements into controllers. We believe, however, that one should be very careful with the use of this property. Most examples in which delay can help in stabilizing closed-loop dynamics can also be easily stabilized using rational controllers. The latter are typically simpler, both conceptually and from an implementation point of view. Moreover, the incorporation of additional loop delays might lead to a deterioration in robustness properties.

2.2 Pole location

Adapting classical pole location methods to dead-time systems is considerably more complicated than adapting the Nyquist arguments. The characteristic polynomial (more precisely, quasi-polynomial) of the closed-loop system in Fig. 2 has the form

$$\chi_{\rm cl}(s) = A(s) + B(s)\,{\rm e}^{-sh}, \tag{1}$$

where the polynomials

$$A(s) = s^n + a_{n-1}s^{n-1} + \cdots + a_0 \quad \text{and} \quad B(s) = b_m s^m + b_{m-1}s^{m-1} + \cdots + b_0$$

are the denominator and numerator, respectively, of the loop transfer function $L_{\rm r}(s)$ defined in §2.1. The characteristic quasi-polynomial (1) is not rational and has an infinite number of roots for every $h > 0$. This fact renders the root locus method useless.

In the stability analysis of (1) the following arguments can be used. The key property of the quasi-polynomial (1) in the case of $n \geq m$ and $|b_n| < 1$ (otherwise the system is unstable for all $h > 0$; see §4.1) is the continuity of its roots as functions of positive h. This means that we can start from the case of $h = 0$, where the stability of (1) can be analyzed using well-understood classical methods, and then count the *imaginary axis crossings* of the roots of (1) as h increases. The roots[2] may cross the axis from left to right (i.e., become unstable), from right to left (i.e., become stable), or just be a point of tangency with the axis. The analysis of the stability of (1) by this approach is simplified owing to the fact that neither the roots at which $j\omega$-axis crossings take place (which are actually the *crossover frequencies* of $L_{\rm r}(j\omega)$) nor the directions of the crossings depend on h. Moreover, the calculations of the crossover frequencies and the crossing directions are based on polynomial equations only.

Indeed, it is readily seen that whenever $A(s)$ and $B(s)$ do not have common imaginary axis roots, $\chi_{\rm cl}(j\omega) = 0$ iff both

$$|A(j\omega)|^2 - |B(j\omega)|^2 = 0 \tag{2a}$$

and

[2]Actually, a pair of roots at each crossing as roots cannot cross the imaginary axis at the origin when h varies. This implies that if $A(s) + B(s)$ has an odd number of unstable roots (equivalently, $A(0) + B(0) < 0$), $\chi_{\rm cl}(s)$ can be stable for no h.

$$\omega h = \arg\Big(-\frac{B(j\omega)}{A(j\omega)}\Big) + 2\pi k \quad \text{for some } k \in \mathbb{Z}^+, \tag{2b}$$

where, with no loss of generality, we assume that $\arg(\cdot) \in [0, 2\pi)$. It is clear, that for any non-zero real solution ω_c of (2a) there always exists an $h > 0$ (actually, a family of delays of the form $h_0 + \frac{2\pi}{\omega_c}k$) such that (2b) holds for $\omega = \omega_c$ as well. Therefore, the crossover frequencies are actually all *positive real* solutions[3] of the polynomial equation (2a), which does not depend on h. Furthermore, it turns out [6] that the direction in which the roots of (1) cross the imaginary axis at $s = j\omega_c$ as h increases depends solely on the sign of the polynomial

$$\sigma(\omega) \doteq \frac{\mathrm{d}}{\mathrm{d}\omega}\big(|A(j\omega)|^2 - |B(j\omega)|^2\big) \tag{3}$$

at the crossover frequencies which, again, is independent of h. If $\sigma(\omega_c) > 0$, a root crosses the axis from left to right (this situation is called a *switch*); if $\sigma(\omega_c) < 0$, a root crosses from right to left (a *reversal*); and, if $\sigma(\omega_c) = 0$, there may not be any $j\omega$-axis crossings (it depends on higher derivatives then).

To illustrate the application of the ideas outlined above to the stability analysis of (1), we again consider the system in Example 1.

Example 2. The characteristic polynomial of the system in Example 1 is

$$\chi_{\mathrm{cl}}(s) = s^2 + 0.1s + 1 + 0.4\,\mathrm{e}^{-sh},$$

so that (2a) and (3) become $\omega^4 - 2 \cdot 0.995\,\omega^2 + 0.84 = 0$ and $\sigma(\omega) = 4\omega(\omega^2 - 0.995)$, respectively. Thus, there exist two crossover frequencies $\omega_{c1} = 0.78$ and $\omega_{c2} = 1.176$ which are a reversal and a switch, respectively ($\sigma(0.78) < 0$ and $\sigma(1.176) > 0$). Furthermore, it follows from (2b) that the delays at which the switches and the reversals occur are

$$h_{\mathrm{switch}} = 0.254 + 5.344\,k \quad \text{and} \quad h_{\mathrm{reversal}} = 3.778 + 8.06\,k,$$

respectively (for $k = 0, 1, \ldots$). Thus, as all roots of $\chi_{\mathrm{cl}}(s)$ for $h = 0$ are stable, the system is stable for $h \in [0, 0.254)$. At $h = 0.254$ the first switch occurs and the system becomes unstable. Yet before the second switch takes place at $h = 5.598$ (when another pair of poles migrates from left to right) we have a reversal at $h = 3.778$, which means that the two poles, that became unstable under the first switch, return to the left half-plane (LHP) again. Therefore, the system is stable in $h \in (3.778, 5.598)$ as well. This stability interval, however, is the last one as the second reversal at $h = 11.839$ occurs after *two* switches at $h = 5.598$ and $h = 10.942$, i.e., at the time of the second reversal there are four poles in the right half-plane (RHP). Since the distance between two subsequent switches is strictly smaller than the distance between two subsequent reversals (generic in the case of second-order $A(s)$ and $B(s)$), more and more roots of (1) are accumulated in the RHP as h increases.

[3] If there are no positive real solutions of (2a), no poles migrate from left to right or vice versa as h varies and the stability (or instability) of the roots of (1) is *delay independent*.

2.3 State space

If the state-space realization of the plant P is given by $\dot{x} = Ax + Bu$, then the presence of an input delay brings it to the form

$$\dot{x}(t) = Ax(t) + Bu(t-h), \tag{4}$$

where $x \in \mathbb{R}^n$ and $u \in \mathbb{R}^m$. The important point here is that x in (4) is no longer a state vector in the sense of Poincaré (history accumulator). That is, a knowledge of $x(t_0)$ and the future inputs is not sufficient to calculate the future evolution of $x(t)$. The information above should be complemented by the knowledge of the input $u(t)$ over the "history window" $[t_0 - h, t_0)$. For this reason, the *complete state vector* of (4) at time t is $(x(t), \breve{u}_t)$, where $\breve{u}_t$ denotes the finite window history of u: $\breve{u}_t(\tau) = u(t+\tau)$, $\tau \in [-h, 0]$.

Writing down the "true state equation" for the system in (4) requires the introduction of some additional technicalities. Going along this line, however, would digress from our main purpose. For that reason, we describe properties of system (4) using analogies from systems operating in discrete time. These systems are finite dimensional and somewhat more intuitive to deal with. On the other hand, the underlying ideas in the discrete- and continuous-time cases are very similar.

The difference equation of the discrete-time version of the plant, $P(z)z^{-h}$, is

$$x(t+1) = Ax(t) + Bu(t-h), \qquad h \in \mathbb{Z}^+. \tag{5}$$

As in the continuous-time case, $x(t)$ can no longer be regarded as its state vector. It should be complemented by the finite history of u. Namely, the history is now accumulated by the $(n + hm)$-dimensional vector

$$\hat{x}(t) \doteq \begin{bmatrix} x'(t) & u'(t-h) & u'(t-h+1) & \dots & u'(t-1) \end{bmatrix}'$$

and the state-space equation of (5) becomes

$$\hat{x}(t+1) = \begin{bmatrix} A & B & 0 & \dots & 0 \\ 0 & 0 & I & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \dots & I \\ 0 & 0 & 0 & \dots & 0 \end{bmatrix} \hat{x}(t) + \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ I \end{bmatrix} u(t) \tag{5_{aug}}$$

which is also referred to as the *augmented form*.

The choice of the state vector and the form of the state equation in (5_{aug}) have far-reaching consequences on the analysis and design of time-delay systems. For example, the notion of static state feedback should be adopted to the changes. Indeed, for system (5) the control law $u(t) = Fx(t)$ is, strictly speaking, a static output feedback for the measurement output $y(t) = \begin{bmatrix} I & 0 & \dots & 0 \end{bmatrix} \hat{x}(t)$. The static state feedback in the time-delay case is the control law of the form

$$u(t) = \begin{bmatrix} F_x & F_{u,1} & \dots & F_{u,h} \end{bmatrix} \hat{x}(t) = F_x\, x(t) + \sum_{i=-h}^{-1} F_{u,h+1+i}\, u(t+i),$$

which is actually a *dynamic feedback* in terms of the state vector x of the delay-free plant $P(z)$. The continuous-time counterpart of the control law above is

$$u(t) = F_x\, x(t) + \int_{-h}^{0} F_u(h+\tau) u(t+\tau) \mathrm{d}\tau \tag{6}$$

for some $m \times n$ matrix F_x and an $m \times m$ valued function $F_u(t)$ defined over the interval $[0, h]$. The control law of the form (6) is infinite dimensional (though implementable) and is called a *distributed-delay control law* due to the form of the second term on its right-hand side. Some aspects of the choice of its parameters F_x and $F_u(t)$ will be discussed in §3.2.

Another example of how the definition of the state space for time-delay systems affects the state-space analysis of the system in Fig. 2 is the Lyapunov (second) method. The Lyapunov method is a powerful tool for the analysis of control systems. The idea, roughly, is to construct a positive function in the state vector (it can be interpreted as a potential function), the derivative of which along the trajectory of the system should be negative to guarantee stability. For example, for finite-dimensional linear time-invariant (LTI) systems a common choice of the Lyapunov function is the quadratic function $V(x) = x'Px$ for some $P = P' > 0$. In the dead-time case such a choice would capture only a part of the state vector. In other words, the positivity requirement would be violated.[4] In this circumstance, a natural choice is a quadratic functional constructed on the infinite-dimensional state vector $(x(t), \breve{u}_t)$, i.e.,

$$V(x) = x'P_{xx}x + 2x' \int_{-h}^{0} P_{xu}(\tau) u(t+\tau) \mathrm{d}\tau + \int_{-h}^{0} \int_{-h}^{0} u'(t+\sigma) P_{uu}(\tau, \sigma) u(t+\tau) \mathrm{d}\sigma \mathrm{d}\tau$$

subject to an appropriate constraint on the positivity of the functional. The functional above is known as the *Krasovskii functional*. For more details see, e.g., [2].

2.4 Rational approximations of delay systems

Many technical difficulties in dealing with time-delay systems in continuous time originate from the infinite dimensionality of the delay element e^{-sh}. Natural questions arising in this respect are whether and how a delay system can be approximated by a finite-dimensional one to which standard analysis and

[4] The function $x'Px$ can still be used under modifications of the conditions on the negativeness of its derivative as, e.g., in the *Razumikhin approach*.

design methods can be applied. Below we outline some ideas used to answer these questions. For a more detailed and rigorous discussion and additional references the reader is referred to the survey paper [7].

To start with, note that the rational approximation of the pure delay, e^{-sh}, is fairly hopeless. This follows from the fact that any rational transfer function can have only a finite phase lag, whereas the phase lag of $\mathrm{e}^{-j\omega h}$ approaches infinity as ω increases. As the magnitude of the frequency response of the delay element is unity at all frequencies, the arbitrarily large high-frequency phase mismatch results in approximation errors of at least 100% at frequencies where the phase error is $(2k+1)\pi$ for some natural k. Thus, the rational approximation of the delay makes sense only when considered over a finite bandwidth or, equivalently, when a transfer function of the form $R(s)\,\mathrm{e}^{-sh}$, for some *strictly proper* $R(s)$, is approximated. In the latter case rational approximations do converge, even when they do not take into account the properties of $R(s)$.

Arguably, the approximation methods most widely used in practice are based on the approximation of the delay element by the ratio

$$\mathrm{e}^{-sh} \approx \frac{Q_n(-s)}{Q_n(s)}, \tag{7}$$

where $Q_n(s)$ is a *stable* polynomial of degree n. For example, Q_n obtained via the $[n,n]$ Padé approximation[5] of e^{-sh} are given by

$$Q_n(s) = \sum_{i=0}^{n} \binom{n}{i} \frac{h^i(2n-i)!}{(2n)!}\, s^i = \sum_{i=0}^{n} \frac{h^i(2n-i)!n!}{(2n)!(n-i)!\,i!}\, s^i.$$

The first two Padé approximations, for instance, are $Q_1(s) = 1 + \frac{sh}{2}$ and $Q_2(s) = 1 + \frac{sh}{2} + \frac{s^2h^2}{12}$. Another possible choice of Q_n is the truncated power series of $\mathrm{e}^{sh/2}$, i.e.,

$$Q_n(s) = \sum_{i=0}^{n} \frac{h^i}{2^i i!}\, s^i,$$

which is motivated by the equality $\mathrm{e}^{-sh} = \mathrm{e}^{-sh/2}/\mathrm{e}^{sh/2}$. The polynomials corresponding to the first two n are now $Q_1(s) = 1 + \frac{sh}{2}$ and $Q_2(s) = 1 + \frac{sh}{2} + \frac{s^2h^2}{8}$ (the latter is related to the Kautz approximation).

Approximations of the form (7) are better in the low-frequency range. A common approach to simplify the approximation formulae is to exploit the equality $\mathrm{e}^{-sh} = (\mathrm{e}^{-sh/m})^m$. The term $\mathrm{e}^{-sh/m}$ can be approximated by a low-order transfer function of the form (7), typically with $n = 1$ or $n = 2$, and then higher-order approximations are obtained by the increase of m.

Potentially, better approximations may be produced when properties of the "weighting" function $R(s)$ are taken into account in the approximation of

[5] In this method the coefficients of $Q_n(s)$ are chosen so that the first $n+2$ coefficients of the power series expansions of both sides of (7) are matched.

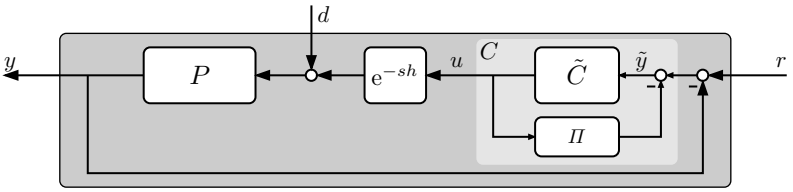


Fig. 5. Dead-time compensator

$R(s)\,\mathrm{e}^{-sh}$. Examples of such an approach are the balanced truncation method and the Hankel norm approximation, see [7]. Yet these methods are considerably more complicated, both computationally and conceptually. For this reason simpler approximation methods, like those described above, are used in most cases.

3 Delay-oriented Design: Dead-Time Compensation

The design of finite-dimensional controllers for (infinite-dimensional) dead-time systems is typically rather conservative. Most available approaches resort to strictly sufficient conditions and result in rather conservative designs. Attractive alternatives in this respect are offered by controllers involving infinite-dimensional dead-time compensators (DTCs). In this section we discuss two (actually equivalent) DTC configurations: the Smith predictor (§3.1) and the finite spectrum assignment (§3.2) control. We also address the DTC version of the two-degree-of-freedom (2DOF) controller configuration (§3.3).

3.1 Smith predictor and its modifications

The classical example of a DTC is the Smith controller [8], which combines clear design guidelines and relatively simple implementation, especially using digital equipment. The block diagram of the Smith controller is depicted in Fig. 5, where the controller C (light-gray box) consists of a *primary controller* $\tilde{C}$ and an internal feedback of the form

$$\Pi(s) = P(s) - P(s)\mathrm{e}^{-sh} \tag{8}$$

called the *Smith predictor*.

The rationale behind the introduction of the infinite-dimensional internal feedback can be appreciated through the examination of the signal $\tilde{y}$, which is the input to the primary controller. It is readily seen that $\tilde{y} = r - Pd - Pu$, which means that the loop $u \circlearrowleft \tilde{y}$ does not contain any delay (i.e., the Smith predictor "compensates" the loop delay). Then, the closed-loop transfer functions are simplified, for example:

$$T_{yr} = \frac{P\tilde{C}}{1+P\tilde{C}}\,\mathrm{e}^{-sh} \quad \text{and} \quad T_{yd} = \frac{1+P\tilde{C}(1-\mathrm{e}^{-sh})}{1+P\tilde{C}}\,P. \tag{9}$$

The delay is thus eliminated from the feedback loop in the sense that it no longer appears in the denominator of the closed-loop transfer functions. The immediate consequence of this fact is that the characteristic equation of the closed-loop system is polynomial, which offers a clear advantage over the quasi-polynomial (1). Thus, provided the plant P is stable, the closed-loop system in Fig. 5 is stable iff $\tilde{C}$ stabilizes the *delay-free* plant P.

This enables one to end up with the following *two-stage design procedure*, which constitutes the core of the dead-time compensation philosophy:

$\boldsymbol{\mathcal{S}_1}$: the design of the primary controller $\tilde{C}$ is based on the delay-free plant P (with additional constraints due to the delay);

$\boldsymbol{\mathcal{S}_2}$: the resulting controller is implemented by adding the Smith predictor as an internal feedback.

Thus, although the overall controller is infinite dimensional, the *design procedure* is completely finite dimensional, so that well-understood control methods can be used for the design of $\tilde{C}$. It is worth stressing that although $\tilde{C}$ in $\boldsymbol{\mathcal{S}_1}$ is designed for the delay-free system, it should not be designed "as if there were no delays" at all [9]. Rather, the loop delay imposes *implicit* constraints on the choice of $\tilde{C}$.

The idea of [8] had a large impact on both the theory and the practice of the control of time-delay systems. Over the years, numerous studies of the properties of the Smith controller and its modifications have been carried out, both in academia and in industry; see the review paper [10] and the references therein. When digital controllers began to appear in the marketplace in the beginning of the 1980s, it became relatively easy to implement the predictor block. As a result, many industrial controllers offer the Smith predictor as a standard algorithm, like the proportional-integral-differential (PID).

A disadvantage of the Smith controller is that it can only be applied to stable plants (as the closed-loop characteristic equation contains all poles of $P(s)$; see [10]). This problem, however, can be overcome by replacing the Smith predictor block in Fig. 5 with

$$\Pi(s) = \tilde{P}(s) - P(s)\mathrm{e}^{-sh}, \tag{10}$$

where $\tilde{P}$ is any rational transfer function such that the resulting Π has no RHP poles. This Π is referred to as the *modified (or generalized) Smith predictor*. It can be shown that in this case C stabilizes the system iff $\tilde{C}$ stabilizes $\tilde{P}$; stage $\boldsymbol{\mathcal{S}_1}$ above should then be replaced with

$\boldsymbol{\mathcal{S}_{1a}}$: the primary controller $\tilde{C}$ is designed for the delay-free *auxiliary* plant $\tilde{P}$ (with additional constraints due to the delay)

rather than for P as in the original Smith controller case.

When P is stable, $\tilde{P}$ can be any stable transfer function. Two particular choices are $\tilde{P} = P$ (resulting in the Smith predictor) and $\tilde{P} = 0$ (which results in the internal model controller configuration [11]). For unstable P the choice of $\tilde{P}$ is less evident, yet the required $\tilde{P}$ can always be found, as illustrated by the following simple example.

Example 3. Let $P = \frac{1}{s-a}$ for some $a \geq 0$. Then a possible choice is $\tilde{P} = \frac{\mathrm{e}^{-ah}}{s-a}$. To see this, note that the pole at $s = a$ is actually canceled in $\Pi(s) = \frac{\mathrm{e}^{-ah} - \mathrm{e}^{-sh}}{s-a}$ by its zero at $s = a$ (it is readily seen that $\Pi(a) = h\mathrm{e}^{-ah}$). It should be stressed that this pole-zero cancellation must be performed before Π is implemented. Otherwise, the uncanceled pole becomes an unstable hidden mode in Π. To see this, assume that Π is implemented by the following equation:

$$\dot{x}(t) = ax(t) + \mathrm{e}^{-ah}u(t) - u(t-h).$$

These dynamics are not internally stable because any error in the computation of $\mathrm{e}^{-ah}u(t) - u(t-h)$ will eventually lead to an unbounded x. On the other hand, assuming zero initial conditions we can write

$$\begin{aligned} x(t) &= \int_0^t \mathrm{e}^{a(t-\tau)}\mathrm{e}^{-ah}u(\tau)\mathrm{d}\tau - \int_h^t \mathrm{e}^{a(t-\tau)}u(\tau-h)\mathrm{d}\tau \\ &= \int_0^t \mathrm{e}^{a(t-h-\tau)}u(\tau)\mathrm{d}\tau - \int_0^{t-h} \mathrm{e}^{a(t-h-\tau)}u(\tau)\mathrm{d}\tau \\ &= \int_{t-h}^t \mathrm{e}^{a(t-h-\tau)}u(\tau)\mathrm{d}\tau = \int_{-h}^0 \mathrm{e}^{-a(h+\tau)}u(t+\tau)\mathrm{d}\tau, \end{aligned}$$

which is a representation of Π after the pole at $s = a$ is canceled and it is stable (it is a finite impulse response system).

In general, when the plant is given in terms of its state-space realization $P(s) = C(sI - A)^{-1}B$ the following choice of $\tilde{P}$ is always admissible:

$$\tilde{P}(s) = C\mathrm{e}^{-Ah}(sI - A)^{-1}B = C(sI - A)^{-1}\mathrm{e}^{-Ah}B \tag{11}$$

as all poles of $P(s)$ are canceled in Π. The corresponding Π should then be implemented as follows:

$$x(t) = C\int_{t-h}^t \mathrm{e}^{A(t-h-\tau)}Bu(\tau)\mathrm{d}\tau = C\int_{-h}^0 \mathrm{e}^{-A(h+\tau)}Bu(t+\tau)\mathrm{d}\tau$$

(see [12] for a discussion on the implementation of the system above and further references).

It is worth emphasizing that stability of the closed-loop system is not the only rationale in the choice of $\tilde{P}$. The latter also affects properties of the resulting control system as the internal feedback Π alters the primary controller $\tilde{C}$. In other words, some properties of the primary controller (designed to work

with $\tilde{P}$) might not be inherited by the overall controller C and the resulting control loop. This situation is confusing, since a "good" design in the stage $\boldsymbol{\mathcal{S}_{1a}}$ might mean nothing in terms of the original system.

The problem and possible remedies can be illustrated by the following example. It is readily seen that the presence of an integrator in $\tilde{C}$ does not necessarily imply that the resulting controller C includes an integral action. Indeed, let $\tilde{C}(s) = \frac{1}{s}\tilde{C}_0(s)$ for some $\tilde{C}_0$ with a bounded static gain $\tilde{C}_0(0)$. Then

$$C(0) = \lim_{s\to 0} \frac{\tilde{C}(s)}{1+\tilde{C}(s)\Pi(s)} = \lim_{s\to 0} \frac{\tilde{C}_0(s)}{s+\tilde{C}_0(s)\Pi(s)} = \frac{1}{\Pi(0)}$$

($\Pi(0)$ is well defined because Π is stable). Thus, the overall controller contains an integral action (has a singularity at the origin) iff $\Pi(0) = 0$, i.e., $\Pi(s)$ has a zero at the origin.[6] This condition can be easily incorporated into the choice of $\tilde{P}$ by, i.e., adding to it an appropriate constant (which clearly does not affect the stability of Π). For example, $\tilde{P}$ in (11) may be replaced by

$$\tilde{P}(s) = C\mathrm{e}^{-Ah}(sI-A)^{-1}B - C\int_0^h \mathrm{e}^{-A\tau}\mathrm{d}\tau B,$$

which guarantees that $\Pi(0) = 0$ (the resulting Π is sometimes referred to as the *Watanabe–Ito predictor*).

The requirement $\Pi(0) = 0$ can be interpreted as the requirement to keep the internal feedback in the controller small at $s = 0$ so that it does not alter the primary controller. This may be naturally extended to the requirement to keep $|\Pi(j\omega)|$ small over a required frequency range, typically in the low-frequency range up to the crossover frequency of the designed loop $\tilde{L} = \tilde{P}\tilde{C}$. In other words, the rationale behind the choice of $\tilde{P}$ may be to seek a "good" approximation of $P\mathrm{e}^{-sh}$ in the required frequency range under a constraint on the stability of the resulting Π. Such an approach potentially has the following two advantages. First, it makes properties of the resulting closed-loop system closer to the properties of the finite-dimensional systems comprised from $\tilde{P}$ and $\tilde{C}$, for which the design in $\boldsymbol{\mathcal{S}_{1a}}$ is performed. For example, it can be shown that $S = (I-\Pi\tilde{C})\tilde{S}$, where $\tilde{S} \doteq 1/(1+\tilde{L})$ and $S \doteq 1/(1+L)$ are the designed and actual sensitivity functions, respectively. Hence, the relative difference between the designed and actual sensitivities, $1 - S/\tilde{S} = \Pi\tilde{C}$, is proportional to Π. Second, when $\tilde{P}$ is close to P, design limitations imposed by the delay (caused by the phase lag of e^{-sh}) do show up in the design of the primary controller. This may prevent an excessively aggressive design of the primary

[6] Although the internal feedback loop in the controller contains then an unstable pole-zero cancellation (at $s = 0$), this cancellation does not give rise to any instability. Intuitively, this follows from the fact that, despite the cancellation, the pole at the origin is not a hidden mode of $C(s)$ from its input $r - y$ to its output u. This can be proved formally by introducing an additional (fictitious) input immediately after Π and verifying stability of all resulting closed-loop systems.

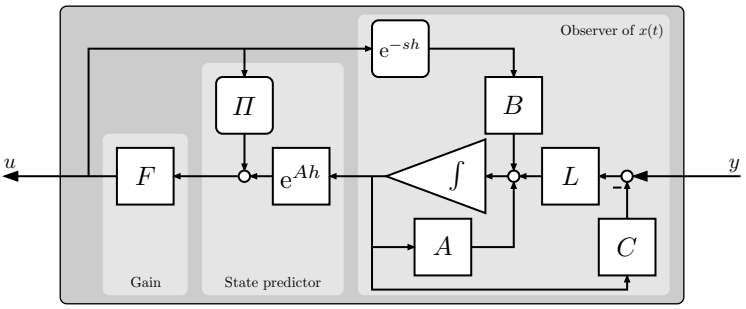


Fig. 6. Observer-predictor

controller (otherwise, one might be tempted to use $\tilde{C}$ which is too aggressive, e.g., it has too high a bandwidth).

3.2 Finite spectrum assignment

Another infinite-dimensional yet implementable structure of DTC can be deduced from the "true" state feedback control law (6). It can be shown [13] that for $F_x = F\mathrm{e}^{Ah}$ and $F_u(t) = F\mathrm{e}^{A(h-t)}B$, i.e., when the control law is

$$u(t) = F\left(\mathrm{e}^{Ah}x(t) + \int_{-h}^{0} \mathrm{e}^{-A\tau}Bu(t+\tau)\mathrm{d}\tau\right), \tag{12}$$

the closed-loop system has only a finite number of finite poles, namely, those satisfying the characteristic equation $\det(sI - A - BF) = 0$. The latter is exactly the characteristic equation of the delay-free system under state feedback control of the form $u = Fx$. Thus, the control law (12) transforms the pole placement problem for a time-delay system to that of its delay-free counterpart. Such control strategy is called the *finite spectrum assignment*.

The control law described above has an interesting interpretation. To see this, note that (12) can be rewritten as $u = Fx_{\mathrm{p}}$, where

$$x_{\mathrm{p}}(t) = \mathrm{e}^{Ah}x(t) + \int_{-h}^{0} \mathrm{e}^{-A\tau}Bu(t+\tau)\mathrm{d}\tau = \mathrm{e}^{Ah}x(t) + \int_{t-h}^{t} \mathrm{e}^{A(t-\tau)}Bu(\tau)\mathrm{d}\tau \tag{13}$$

actually coincides with $x(t+h)$ calculated according to (4). In other words, x_{p} is the h time units ahead *prediction* of x. This implies that the control law (12) is a *state predictor controller*. The prediction here compensates, in a sense, for the input delay, resulting in a "delay-free" system. The latter is then controlled by a standard static state feedback.

The use of the predictor in (12) has some resemblance with the observer-based control. In both cases the missing states are reconstructed (by either the observer or the predictor) and then used by the controller as if they were the true states. It is then not a surprise that state prediction and observation can

be combined when the required state, $x(t+h)$, is *both* delayed *and* partially unknown (i.e., only its part, Cx, is measured). The resulting scheme, known as the *observer-predictor*, is depicted in Fig. 6. The observer-predictor consists of the standard (Luenberger) observer of $x(t)$ with a gain L, the predictor block which contains a distributed-delay element Π generating the second term on the right-hand side of (13), and the static state-feedback gain F. It can be shown [14] that the resulting closed-loop system also has a finite number of finite poles, namely those satisfying $\det(sI - A - BF)\det(sI - A - LC) = 0$, exactly as in the delay-free case.

Curiously, the observer-predictor and the modified Smith predictor controllers were regarded as different configurations of DTC for almost two decades. It turns out, however, that these schemes are equivalent modulo a simple state transformation [15]. More precisely, the observer-predictor controller is equivalent to the modified Smith predictor when $\tilde{P}$ is chosen as in (11) and the primary controller $\tilde{C}$ is the standard observer-based feedback law for this $\tilde{P}$. The equivalence, however, is valid only under the ideal implementation of the distributed-delay block Π. When the latter is approximated, the modified Smith predictor has some advantages over the observer-predictor [12].

3.3 2DOF dead-time compensation

In many applications control objectives include both disturbance (load) attenuation and reference (command or set-point) tracking. In cases where the disturbance can be measured, the predictor in the DTC can be used effectively to handle tracking and disturbance attenuation simultaneously, see [16]. In the other circumstance, these objectives can be handled independently by the use of two-degree-of-freedom (2DOF) controller configurations. The idea is to use the feedback component of the controller to attenuate *unmeasured* disturbances and cope with modeling uncertainty and the feedforward part of the controller (prefilter) to obtain the desired response to *measured* (command) signals.

Below we show how this idea can be exploited in the case of dead-time systems. To simplify the exposition and avoid the introduction of the coprime factorization machinery, we consider only stable $P(s)$ (the general case is addressed in [17]). The 2DOF modified Smith predictor for such systems is shown in Fig. 7, where $\tilde{C}(s)$ is the (stabilizing) primary feedback controller and $K(s)$ is the prefilter. The central point here is that the primary controller is split in two parts as

$$\tilde{C} = (1 + \tilde{P}\tilde{C})\,\frac{\tilde{C}}{1+\tilde{P}\tilde{C}}$$

and the feedforward component enters the loop in between these parts. From a closed-loop performance point of view, this partition of $\tilde{C}$ does not change

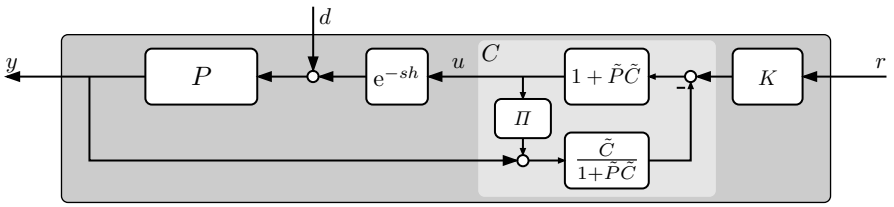


Fig. 7. 2DOF dead-time compensator

anything[7] and $\tilde{C}$ can be designed exactly as in the 1DOF case (K obviously does not affect the closed-loop properties). The way in which the feedforward component enters the loop aims to decouple the feedforward properties from the effect of $\tilde{C}$. It can be verified that the closed-loop transfer functions for the system in Fig. 7 are

$$T_{yr} = PK\,\mathrm{e}^{-sh} \quad \text{and} \quad T_{yd} = \frac{1 + \Pi\tilde{C}}{1 + \tilde{P}\tilde{C}}\,P,$$

so that the effects of r and d are completely decoupled. In other words, $\tilde{C}$ and K can be designed *independently*. Moreover, it is worth stressing that T_{yr} does not depend on the choice of the auxiliary plant $\tilde{P}$.

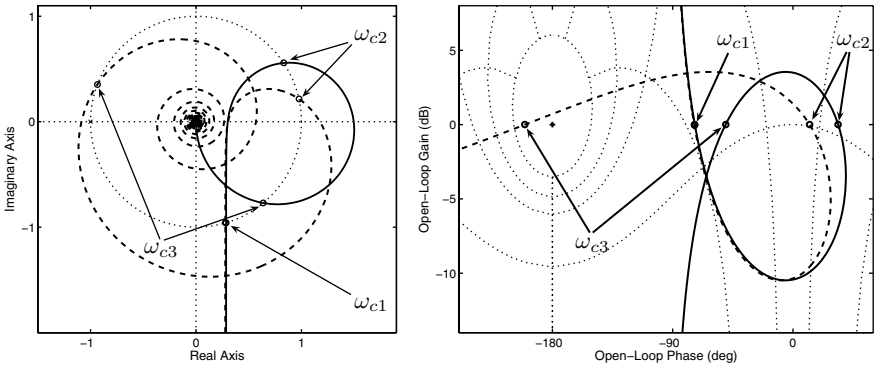
4 Robustness with Respect to Delay Uncertainty/Variation

In many situations, and particularly, in most networked control systems, the loop delay is not a fixed known value but rather is uncertain and/or slowly varying. If this is the case, one should be concerned not only with the stability for a nominal value of h, but also with the sensitivity of the closed-loop stability to changes in the delay. In this section we address some basic ideas of how delay uncertainty can be handled. In particular, we discuss the notion of the delay margin (§4.1) and some underlying ideas that can be used to simplify the analysis of uncertain time delays (§4.2).

4.1 Delay margin (dead-time tolerance)

Although loop delays affect only the phase of the open-loop transfer function, the phase margin μ_{ph} might be a poor measure of the robustness against delay variations. The reason is that the phase lag due to the delay depends on the crossover frequency ω_c at which μ_{ph} is measured, whereas the phase margin

[7]Note that $\tilde{P}$ must be stable since otherwise $\Pi = \tilde{P} - P\mathrm{e}^{-sh}$ cannot be stable, and $\frac{1}{1+\tilde{P}\tilde{C}}$ and $\frac{\tilde{C}}{1+\tilde{P}\tilde{C}}$ must be stable because $\tilde{C}$ is stabilizing. These facts guarantee that there are no unstable pole-zero cancellations between the two parts above.



(a) Nyquist diagram (b) Nichols chart

Fig. 8. Plant with several crossover frequencies: $L_{\rm r}(s)$ (solid line) and $L_{\rm r}(s)\mathrm{e}^{-0.5s}$ (dashed line)

notion does not take the crossover frequency into account. For example, the systems with $L = 2/(10s + 1)$ and $L = 2/(0.1s + 1)$ have identical phase margins (their Nyquist plots coincide), yet the former is stable for all delays up to 12.1, whereas the latter—only up to 0.121. This calls for the introduction of a new stability margin, called the *delay margin* or *dead-time tolerance*. The delay margin, $\mu_{\rm d}$, is defined as *the smallest additional delay destabilizing the system*.

It is somehow conventional in the control literature to refer to the delay margin as the following quantity:

$$\mu_{\rm d} = \frac{\mu_{\rm ph}}{\omega_c}. \tag{14}$$

Yet this is correct only when the system has just one crossover frequency and the high-frequency gain is smaller than 1. When the latter condition is not satisfied, e.g., for $L_{\rm r}(s) = \frac{2s+1}{s+2}$, the delay margin is zero since any loop delay destabilizes the system. Intuitively, this can be seen from the fact that the Nyquist plot of non-strictly proper dead-time systems at high frequencies encircles the origin an infinite number of times along a circle, the radius of which is the loop high-frequency gain. When the latter is larger than one, these encirclements include the critical point as well.

When there are several crossover frequencies, (14) also falls short of reflecting $\mu_{\rm d}$. To see this, consider the following loop transfer function:

$$L_{\rm r}(s) = \frac{6(s^2 + 0.2s + 0.01)}{s(s + 2)^2}.$$

The Nyquist and Nichols plots of this system are presented in Fig. 8 by the solid lines. This system has three crossover frequencies: $\omega_{c1} = 0.0154$, $\omega_{c2} = 0.746$, and $\omega_{c3} = 5.24$. The phase margin here is measured at ω_{c1}, so that

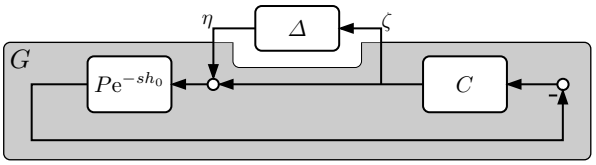


Fig. 9. Robust framework

the delay margin calculated according to (14) is $\mu_{\rm d} > \frac{\pi}{2\cdot 0.0154} \approx 102$. Yet this conclusion is erroneous. This is clearly seen in the frequency-response plots of $L_{\rm r}(s)e^{-0.5s}$ (dashed lines in Fig. 8). These curves do encircle the critical point, i.e., the destabilizing delay is actually < 0.5 sec. The reason is that the phase lag due to the delay at the largest crossover frequency, ω_{c3}, is much larger than that at ω_{c1}. Hence, when the delay is increased $L_{\rm r}(j\omega_{c3})$ reaches the critical point long before $L_{\rm r}(j\omega_{c1})$ does, even though the phase distance from the critical point in the latter case is smaller.

Following these arguments, the correct formula for the $\mu_{\rm d}$ should be

$$\mu_{\rm d} = \begin{cases} \min\limits_i \dfrac{\mu_{{\rm ph},i}}{\omega_{ci}} & \text{if } \lim_{\omega\to\infty}|L_{\rm r}(j\omega)| < 1 \\ 0 & \text{otherwise,} \end{cases} \tag{15}$$

where ω_{ci} are crossover frequencies and $\mu_{{\rm ph},i}$ are the corresponding "phase margins" (angular distances of $L_{\rm r}(j\omega_{ci})$ to the critical point).

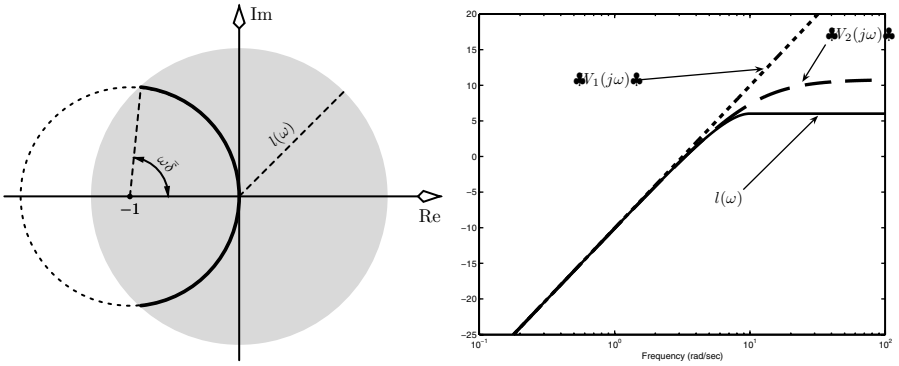
4.2 Unstructured uncertainty embedding

Although the delay margin notion does reflect the sensitivity of the closed-loop system to delay uncertainty, it is not readily incorporated into analytic design procedures. In this section we discuss a possible alternative, which can be roughly classified as embedding delay uncertainty/variations into the finite-dimensional robust stability formalism. The idea is to "cover" the (uncertain) delay element by complex plant perturbations which, in turn, could be handled using standard tools of robust control.

Consider the dead-time system in Fig. 2 and assume that the delay $h = h_0 + \delta$, where $h_0 \geq 0$ is its "nominal value" and δ is an unknown constant satisfying $|\delta| < \bar{\delta}$ for some $\bar{\delta} \leq h_0$. This system can always be presented in the form depicted in Fig. 9 (as we only discuss the stability of the closed-loop system here, we assume that all external signals are zero) with $\Delta(s) = e^{-s\delta} - 1$. Denote the system from η to ζ (the one in the gray box) by G, so that

$$G(s) = \frac{P(s)C(s)e^{-sh_0}}{1 + P(s)C(s)e^{-sh_0}},$$

which is actually the transfer function from r to y for the system in Fig. 2 (complementary sensitivity function) with the nominal delay $h = h_0$. Naturally, we assume hereafter that G is stable. The closed-loop system can then



(a) $e^{-j\omega\delta} - 1$ and its covering disk (b) $l(\omega)$ and its approximations

Fig. 10. Delay uncertainty region

be thought of as the feedback interconnection of two stable[8] systems G and Δ, where the former is known whereas the latter is uncertain.

By *the zero exclusion principle* [2] the closed-loop system in Fig. 9 is stable iff $\Delta(j\omega)G(j\omega) \neq 1$ for all frequencies ω and all δ in the given region. The application of this criterion to the stability analysis of this system is complicated because Δ is highly structured with quite a complicated dependence on the uncertain parameter δ. Indeed, at each ω the only available information about $\Delta(j\omega)$ is that it lies on the arc of a unit radius and a central angle of $2\omega\bar{\delta}$ radians centered at -1. This is shown in Fig. 10(a) by the bold solid curve (at $\omega = \pi/\bar{\delta}$ this arc is closed up to become a shifted unit circle and remains that circle for all larger frequencies). This information is not readily suitable to be incorporated into analytical methods.

A possible way to circumvent this difficulty is to embed the family of systems $e^{-s\delta} - 1$, $|\delta| < \bar{\delta}$, into a larger yet less structured and more manageable family of systems. An example of such a larger family is the smallest disk centered at the origin that contains the arc above; see the gray disk in Fig. 10(a). Denote the radius of this disk, which is a function of the frequency ω, by $l(\omega)$. Elementary geometry yields that

$$l(\omega) = \begin{cases} 2\sin(\frac{\bar{\delta}}{2}\omega) & \text{if } \omega \leq \pi/\bar{\delta}, \\ 2 & \text{otherwise,} \end{cases}$$

which is shown in Fig. 10(b) (solid line). As expected, the radius is small at low frequencies and grows as ω increases. This reflects the fact that the loop delay affects the closed-loop dynamics mostly at high frequencies. Thus, the original problem reduces to the stability problem for all stable $\Delta(s)$ satisfying

[8]Strictly speaking, Δ is not stable as it might include a preview term (negative delay). Yet G includes the delay e^{-sh_0} which can always be augmented to Δ thus making it stable. For that reason, we proceed assuming that Δ *is* stable.

$|\Delta(j\omega)| < l(\omega)$ (this set is called *unstructured uncertainty*). Clearly, the stability of the system with this set of Δ's guarantees the stability of the original system. The opposite, however, is not true: the original system might still be stable even when the unstructured uncertainty destabilizes G. In other words, the embedding procedure introduces *conservatism*. In return, solvability conditions are considerably simplified: the system in Fig. 9 with unstructured uncertainty is stable iff $|G(j\omega)| < 1/l(j\omega)$ for all ω.

The latter is actually the condition on the weighted H^∞ norm of $G(s)$, so that the standard H^∞ machinery can be used to solve the robustness problem. Let us just emphasize the following two aspects of the resulting problem.

- The direct solution of the associated H^∞ problem is complicated due to the fact that the weighting function $l(\omega)$ is irrational. Yet this difficulty can be overcome by covering $l(\omega)$ by the frequency response gain $|W(j\omega)|$ of a rational transfer function $W(s)$. Some possible choices of $W(s)$ are

$$W_1(s) = \bar{\delta}s \quad \text{and} \quad W_2(s) = \frac{3.465\bar{\delta}s}{\bar{\delta}s + 3.465},$$

  which are shown in Fig. 10(b) by dashed lines. This step might introduce some additional conservatism. Yet this conservatism becomes negligible as the order of the rational approximation increases. For example, the magnitude plot of

$$W_3(s) = \frac{2\bar{\delta}s\,(s^2 + 1.676\omega_0 s + \omega_0^2)}{(\bar{\delta}s + 2)(s^2 + 1.370\omega_0 s + \omega_0^2)}, \quad \text{where } \omega_0 = 2.363/\bar{\delta},$$

  practically coincides with that of $l(\omega)$. For a more detailed discussion see [18].
- Another point that might give rise to some concerns is the infinite-dimensionality of $G(s)$ due to the presence of the delay. This might appear to conflict with the goal to end up with a finite-dimensional problem and even might suggest the choice $h_0 = 0$ and the corresponding delay bound $0 \le \delta \le 2\bar{\delta}$ [18] that introduces extra conservatism. The infinite-dimensionality, however, can easily be resolved by the use of the DTC controller. For example, the standard Smith predictor brings G to the form of T_{yr} in (9). The latter contains the delay element only in its numerator and this delay does not affect $|G(j\omega)|$. Thus, the robust stability problem is reduced to a purely finite-dimensional H^∞ problem, the solution of which is well understood. For more details see [15] and the references therein.

It is worth emphasizing that the conversion to the H^∞ optimization problem enables one not only to analyze the stability of the dead-time system for a given controller but also to design C to, e.g., maximize the (upper bound on the) delay margin.

Note also that the reasonings above can be extended to more complicated situations. For example, the multiple delay case can be handled in the μ

framework by similar arguments [2, 19]. Another example is the case of time-varying delay; see [2, 20].

Acknowledgement. This work was supported by the Israel Science Foundation (grant No. 298/04) and the Israel Electric Company (grant No. 793-2000-3).

References

1. J. E. Marshall, H. Górecki, A. Korytowski, and K. Walton, *Time-Delay Systems: Stability and Performance Criteria with Applications*. London: Ellis Horwood, 1992.
2. K. Gu, V. L. Kharitonov, and J. Chen, *Stability of Time-Delay Systems*. Boston: Birkhäuser, 2003.
3. K. Gu and S.-I. Niculescu, Survey on recent results in the stability and control of time-delay systems, *ASME J. Dynamic Systems, Measurement, and Control*, 125(2):158–165, 2003.
4. J.-P. Richard, Time-delay systems: An overview of some recent advances and open problems, *Automatica*, 39(10):1667–1694, 2003.
5. C. A. Desoer and M. Vidyasagar, *Feedback Systems: Input-Output Properties*. New York: Academic Press, 1975.
6. F. J. Boese, Stability with respect to the delay: On a paper of K. L. Cooke and P. van den Driessche, *J. Math. Anal. Appl.*, 228:293–321, 1998.
7. J. R. Partington, Some frequency-domain approaches to the model reduction of delay systems, *Annual Reviews in Control*, 28(1):65–73, 2004.
8. O. J. M. Smith, Closer control of loops with dead time, *Chem. Eng. Progress*, 53(5):217–219, 1957.
9. Z. J. Palmor, Stability properties of Smith dead-time compensator controller, *Int. J. Control*, 32:937–949, 1980.
10. Z. J. Palmor, Time-delay compensation — Smith predictor and its modifications, In *The Control Handbook*, W. S. Levine, Ed., p. 224–237, Boca Raton, FL: CRC Press 1996.
11. M. Morari and E. Zafiriou, *Robust Process Control*. Englewood Cliffs, NJ: Prentice-Hall, 1989.
12. L. Mirkin, On the approximation of distributed-delay control laws, *Syst. Control Lett.*, 55(5):331–342, 2004.
13. A. Z. Manitius and A. W. Olbrot, Finite spectrum assignment problem for systems with delay, *IEEE Trans. Automat. Control*, 24:541–553, 1979.
14. T. Furukawa and E. Shimemura, Predictive control for systems with time delay, *Int. J. Control*, 37(2):399–412, 1983.
15. L. Mirkin and N. Raskin, Every stabilizing dead-time controller has an observer-predictor-based structure, *Automatica*, 39(10):1747–1754, 2003.
16. Z. J. Palmor and D. W. Powers, Improved dead-time compensator controllers, *AIChE Journal*, 31(2):215–221, 1985.
17. L. Mirkin and Q.-C. Zhong, 2DOF controller parametrization for systems with a single I/O delay, *IEEE Trans. Automat. Control*, 48(11):1999–2004, 2003.
18. Z. Q. Wang, P. Lundström, and S. Skogestad, Representation of uncertain time delays in the H^∞ framework, *Int. J. Control*, 59(3):627–638, 1994.

19. Y.-P. Huang and K. Zhou, Robust stability of uncertain time-delay systems, *IEEE Trans. Automat. Control*, 45(11):2169–2173, 2000.
20. C.-Y. Kao and B. Lincoln, Simple stability criteria for systems with time-varying delays, *Automatica*, 40(8):1429–1434, 2004.

Part V

Networking

Network Protocols for Networked Control Systems

F.-L. Lian,[1] J. R. Moyne,[2] and D. M. Tilbury[2]

[1] Electrical Engineering Department, National Taiwan University, No. 1, Sec. 4, Roosevelt Road, Taipei, 106, Taiwan fengli@ntu.edu.tw
[2] Mechanical Engineering Department, University of Michigan, 2350 Hayward St., Ann Arbor, MI 48103, U.S.A. {moyne,tilbury}@umich.edu

1 Introduction

Control systems with networked communication, called *networked control systems* (NCSs), provide several advantages over point-to-point wired systems such as improvement in reliability through reduced volume of wiring, simpler systems integration, easier troubleshooting and maintenance, and the possibility for distributed processing. There are two types of communication networks. *Data networks* are characterized by large data packets, relatively infrequent bursty transmission, and high data rates; they generally do not have hard real-time constraints. *Control networks*, in contrast, must shuttle countless small but frequent packets among a relatively large set of nodes to meet the time-critical requirements. The key element that distinguishes control networks from data networks is the capability to support real-time or time-critical applications [19].

The change of communication architecture from point-to-point to common-bus, however, introduces different forms of time delay uncertainties between sensors, actuators, and controllers. These time delays come from the time sharing of the communication medium as well as the extra time required for physical signal coding and communication processing. The characteristics of time delays may be constant, bounded, or even random, depending on the network protocols adopted and the chosen hardware. This type of time delay could potentially degrade a system's performance and possibly cause system instability.

Thus, the disadvantages of an NCS include the limited bandwidth for communication and the delays that occur when sending messages over a network. In this chapter, we discuss the sources of delay in common communication networks used for control systems, and show how they can be computed and analyzed.

Several factors affect the availability and utilization of the network bandwidth: the sampling rates at which the various devices send information over

the network, the number of elements that require synchronous operation, the method of synchronization between requesters and providers (such as polling), the data or message size of the information, physical factors such as network length, and the medium access control sublayer protocol that controls the information transmission [7]. There are three main types of medium access control used in control networks: random access with retransmission when collisions occur (e.g., Ethernet and most wireless mechanisms), time-division multiplexing (such as master-slave or token-passing), and random access with prioritization for collision avoidance (e.g., Controller Area Network (CAN)). Within each of these three categories, there are numerous network protocols that have been defined and used. For each type of protocol, we study the key parameters of the corresponding network when used in a control situation, including network utilization, magnitude of the expected time delay, and characteristics of time delays. Simulation results are presented for several different scenarios, and the advantages and disadvantages of each network type are summarized. The focus is on one of the most common protocols in each category; the analysis for other protocols in the same category can be addressed in a similar fashion.

A survey of the types of control networks used in industry shows a wide variety of networks in use; see Table 1. The networks are classified according to type: random access (RA) with collision detection (CD) or collision avoidance (CA), or time-division multiplexed (TDM) using token-passing (TP) or master-slave (MS).

Table 1. Worldwide most popular fieldbuses [18]. Note that the totals are more than 100% because many companies use more than one type of bus. Wireless was not included in the original survey, but its usage is growing quickly.

Network	Type	Users	Application domain
Ethernet	RA/CD	50%	Various
Profibus	TDM/(TP and MS)	26%	Process control
CAN-based	RA/CA	25%	Automotive, process
Modbus	TDM/MS	22%	Various
ControlNet	TDM/TP	14%	Plant bus
ASI	TDM/MS	9%	Building systems
Interbus-S	TDM/MS	7%	Manufacturing
Fieldbus Foundation	TDM/TP	7%	Chemical industry
Wireless (e.g., IEEE 802.11)	RA/CA	Unknown	Various

2 Control Network Basics

In this section, we discuss the medium access control (MAC) sublayer protocol of three types of control networks. We focus our discussion on one of the common networks of each type: Ethernet (including hub, switch, and wireless varieties, which will be defined later), ControlNet (a token-passing network), and DeviceNet (a CAN-based network).[3] The MAC sublayer protocol, which describes the protocol for obtaining access to the network, is responsible for satisfying the time-critical/real-time response requirement over the network and for the quality and reliability of the communication between network nodes [8]. Our discussion and comparison thus focus on the MAC sublayer protocols.

For control network operation, the message connection type must be specified. Practically, there are three types of message connections: strobe, poll, and change of state (COS)/cyclic. In a *strobe* connection, the master device broadcasts a strobed message to a group of devices and these devices respond with their current condition. In this case, all devices are considered to sample new information at the same time. In a *poll* connection, the master sends individual messages to the polled devices and requests update information from them. Devices only respond with new signals after they have received a poll message. *COS/cyclic* devices send out messages either when their status is changed (COS) or periodically (cyclic). Although the COS/cyclic connection seems most appropriate from a traditional control systems point of view, strobe and poll are commonly used in industrial control networks [4].

2.1 Ethernet networks (CSMA)

Ethernet generally uses the carrier sense multiple access (CSMA) with CD or CA mechanisms for resolving contention on the communication medium. There are three common flavors of Ethernet: (1) hub-based Ethernet, which is common in office environments and is the most widely implemented form of Ethernet, (2) switched Ethernet, which is more common in manufacturing and control environments, and (3) wireless Ethernet.

Hub-based Ethernet (CSMA/CD)

Hub-based Ethernet uses hub(s) to interconnect the devices on a network. When a packet comes into one hub interface, the hub simply broadcasts the packet to all other hub interfaces. Hence, all of the devices on the same network

[3]Note that Ethernet is not a complete protocol solution but only a MAC sublayer definition, whereas ControlNet and DeviceNet are complete protocol solutions. Following popular usage, we use the term "Ethernet" to refer to Ethernet-based complete network solutions. These include industrial Ethernet solutions such as Modbus/TCP, PROFINET, and EtherNet/IP.

receive the same packet simultaneously, and message collisions are possible. Collisions are dealt with utilizing the CSMA/CD protocol as specified in the IEEE 802.3 network standard [1, 2, 21].

This protocol operates as follows: when a node wants to transmit, it listens to the network. If the network is busy, the node waits until the network is idle; otherwise it transmits immediately. If two or more nodes listen to the idle network and decide to transmit simultaneously, the messages of these transmitting nodes collide and the messages are corrupted. While transmitting, a node must also listen to detect a message collision. On detecting a collision between two or more messages, a transmitting node stops transmitting and waits a random length of time to retry its transmission. This random time is determined by the standard binary exponential backoff (BEB) algorithm: the retransmission time is randomly chosen between 0 and $(2^i - 1)$ slot times, where i denotes the ith collision event detected by the node and one slot time is the minimum time needed for a round-trip transmission. However, after 10 collisions have been reached, the interval is fixed at a maximum of 1023 slots. After 16 collisions, the node stops attempting to transmit and reports failure back to the node microprocessor. Further recovery may be attempted in higher layers [21].

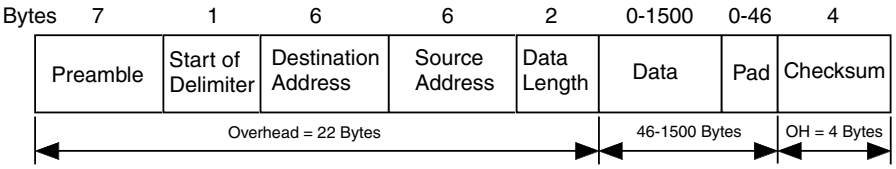


Fig. 1. Ethernet (CSMA/CD) frame format

The Ethernet frame format is shown in Fig. 1 [21]. The total overhead is 26 (=22+4) bytes. The data packet frame size is between 46 and 1500 bytes. There is a nonzero minimum data size requirement because the standard states that valid frames must be at least 64 bytes long, from destination address to checksum (72 bytes including preamble and start of delimiter). If the data portion of a frame is less than 46 bytes, the pad field is used to fill out the frame to the minimum size. There are two reasons for this minimum size limitation. First, it makes it easier to distinguish valid frames from "garbage." When a transceiver detects a collision, it truncates the current frame, which means that stray bits and pieces of frames frequently appear on the cable. Second, it prevents a node from completing the transmission of a short frame before the first bit has reached the far end of the cable, where it may collide with another frame. For a 10-Mbps Ethernet with a maximum length of 2500 m and four repeaters, the minimum allowed frame time or slot time is 51.2 μs, which is the time required to transmit 64 bytes at 10 Mbps [21].

Advantages: Because of low medium access overhead, Ethernet uses a simple algorithm for operation of the network and has almost no delay at low network loads [24]. No communication bandwidth is used to gain access to the network compared with the token bus or token ring protocol. Ethernet used as a control network commonly uses the 10 Mbps standard (e.g., Modbus/TCP); high-speed (100 Mbps or even 1 Gbps) Ethernet is mainly used in data networks [21].
Disadvantages: Ethernet is a nondeterministic protocol and does not support any message prioritization. At high network loads, message collisions are a major problem because they greatly affect data throughput and time delays may become unbounded [24]. The Ethernet capture effect existing in the standard BEB algorithm, in which a node transmits packets exclusively for a prolonged time despite other nodes waiting for medium access, causes unfairness and substantial performance degradation [20]. Based on the BEB algorithm, a message may be discarded after a series of collisions; therefore, end-to-end communication is not guaranteed. Because of the required minimum valid frame size, Ethernet uses a large message size to transmit a small amount of data.

Several solutions have been proposed for using this form of Ethernet in control applications. For example, every message could be time-stamped before it is sent. This requires clock synchronization, however, which is not easy to accomplish, especially with a network of this type [6]. Various schemes based on deterministic retransmission delays for the collided packets of a CSMA/CD protocol result in an upper-bounded delay for all the transmitted packets. However, this is achieved at the expense of inferior performance to CSMA/CD at low to moderate channel utilization in terms of delay throughput [8]. Other solutions also try to prioritize CSMA/CD (e.g., LonWorks) to improve the response time of critical packets [14]. To a large extent these solutions have been rendered moot with the proliferation of switched Ethernet as described below. On the other hand, many of the same issues reappear with the migration to wireless Ethernet for control.

Switched Ethernet (CSMA/CA)

Switched Ethernet utilizes switches to subdivide the network architecture, thereby avoiding collisions, increasing network efficiency, and improving determinism. It is widely used in manufacturing applications. The main difference between switch-based and hub-based Ethernet networks is the intelligence of forwarding packets. Hubs simply pass on incoming traffic from any port to all other ports, whereas switches learn the topology of the network and forward packets to the destination port only. In a star-like network layout, every node is connected with a single cable to the switch. Thus, collisions can no longer occur on any network cable.

Switches employ the cut-through or store-and-forward technique to forward packets from one port to another, using per-port buffers for packets

waiting to be sent on that port. Switches with cut-through first read the MAC address and then forward the packet to the destination port according to the MAC address of the destination and the forwarding table on the switch. On the other hand, switches with store-and-forward examine the complete packet first. Using the cyclic redundancy check (CRC) code, the switch will first verify that the frame has been correctly transmitted before forwarding the packet to the destination port. If there is an error, the frame will be discarded. Store-and-forward switches are slower, but will not forward any corrupted packets.

Although there are no message collisions on the networks, congestion may occur inside the switch when one port suddenly receives a large number of packets from the other ports. Three main queuing principles are implemented inside the switch in this case. They are first-in-first-out (FIFO) queue, priority queue, and per-flow queue. The FIFO queue is a traditional method that is fair and simple. However, if the network traffic is heavy, the quality of service cannot be guaranteed. In the priority queueing scheme, the network manager reads some of the data frames to distinguish which queues will be more important. Hence, the packets can be classified into different levels of queues. Queues with high priority will be processed first followed by queues with low priority until the buffer is empty. With the per-flow queueing operation, queues are assigned different levels of priority (or weights). All queues are then processed one by one according to priority; thus, the queues with higher priority will generally have higher performance and could potentially block queues with lower priority.

Examples of timing analysis and performance evaluation of switched Ethernet can be found in [9, 23].

Wireless Ethernet (CSMA/CA)

Wireless Ethernet, based on the IEEE 802.11 standard, can replace wired Ethernet in a transparent way since it implements the two lowest layers of the International Standards Organization (ISO)/Open Systems Interconnection (OSI) model. Besides the physical layer, the biggest difference between 802.11 and 802.3 is in the medium access control. Unlike wired Ethernet nodes, wireless stations cannot "hear" a collision. A collision avoidance mechanism is used but cannot entirely prevent collisions. Thus, after a packet has been successfully received by its destination node, the receiver sends a short acknowledgment packet (ACK) back to the original sender. If the sender does not receive an ACK packet, it assumes that the transmission was unsuccessful and retransmits.

The collision avoidance mechanism in 802.11 works as follows. If a network node wants to send while the network is busy, it sets its backoff counter to a randomly chosen value. Once the network is idle, the node waits first for an interframe space and then for this backoff time before attempting to send. If another node accesses the network during that time, it must wait again for

another idle interval. In this way, the node with the lowest backoff time sends first. Certain messages (e.g., ACK) may start transmitting after a shorter interframe space, thus they have a higher priority. Collisions may still occur because of the random nature of the backoff time; it is possible for two nodes to have the same backoff time.

Several refinements to the protocol also exist. Nodes may reserve the network either by sending a request to send (RTS) message or by breaking a large message into many smaller messages (fragmentation); each successive message can be sent after the smallest interframe time. If there is a single master node on the network, the master can poll all the nodes and effectively create a TDM contention-free network.

2.2 TDM networks

Time-division multiplexing can be accomplished in one of two ways. In a master-slave network, a single master polls multiple slaves. Slaves can only send data over the network when requested by the master. In this way, there are no collisions, since the data transmissions are carefully scheduled by the master. In a token-passing network, there are multiple masters, or peers. The token bus protocol (e.g., IEEE 802.4) allows a linear, multidrop, tree-shaped, or segmented topology [24]. The node that currently has the token is allowed to send data. When it is finished sending data, or the maximum token holding time has expired, it "passes" the token to the next logical node on the network. If a node has no message to send, it just passes the token to the successor node. The physical location of the successor is not important because the token is sent to the logical neighbor. Collision of data frames does not occur, as only one node can transmit at a time. Most token-passing protocols guarantee a maximum time between network accesses for each node, and most also have provisions to regenerate the token if the token holder stops transmitting and does not pass the token to its successor. In many cases, nodes can also be added dynamically to the bus and can request to be dropped from the logical ring.

ASI, Bitbus, and Interbus-S are typical examples of master-slave networks, while Profibus and ControlNet are typical examples of token-passing networks. Each peer node in a Profibus network can also behave like a master and communicate with a set of slave nodes during the time it holds the token. These are deterministic networks because the maximum waiting time before sending a message frame can be characterized by the token rotation time. The nodes in the token bus network are arranged logically into a ring, and, in the case of ControlNet, each node knows the address of its predecessor and its successor. During operation of the network, the node with the token transmits data frames until either it runs out of data frames to transmit or the time it has held the token reaches the limit. The node then regenerates the token and transmits it to its logical successor on the network.

ControlNet

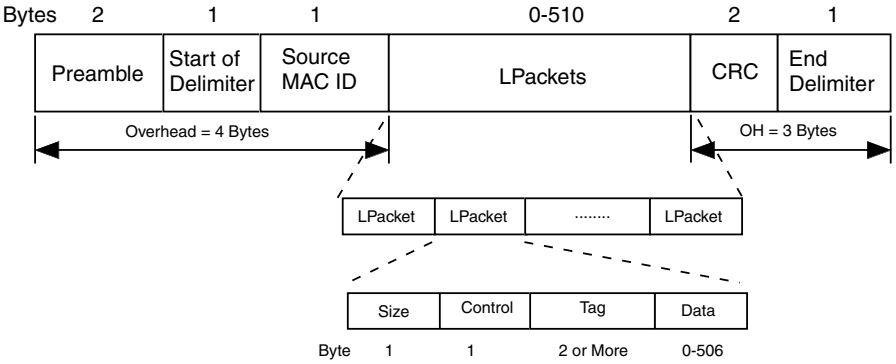


Fig. 2. The message frame of ControlNet (token bus)

The ControlNet protocol is used here as a case study of the operation of a typical token-passing network. The message frame format of ControlNet is shown in Fig. 2 [3]. The total overhead is 7 bytes, including preamble, start delimiter, source MAC ID, CRC, and end delimiter. The data packet frame, also called the link packet (Lpacket) frame, may include several Lpackets that contain size, control, tag, and data fields with total frame size between 0 and 510 bytes. The individual destination address is specified within the tag field. The size field specifies the number of byte pairs (from 3 to 255) contained in an individual Lpacket, including the size, control, tag, and link data fields.

The ControlNet protocol adopts an implicit token-passing mechanism and assigns a unique MAC ID (from 1 to 99) to each node. As in general token-passing buses, the node with the token can send data; however, there is no real token passing around the network. Instead, each node monitors the source MAC ID of each message frame received. At the end of a message frame, each node sets an "implicit token register" to the received source MAC ID + 1. If the implicit token register is equal to the node's own MAC ID, that node may now transmit messages. All nodes have the same value in their implicit token registers, preventing collisions on the medium. If a node has no data to send, it just sends a message with an empty Lpacket field, called a null frame.

The length of a cycle, called the network update time (NUT) in ControlNet or the token rotation time (TRT) in general, is divided into three major parts: scheduled, unscheduled, and guardband, as shown in Fig. 3. During the scheduled part of an NUT, each node can transmit time-critical/scheduled data by obtaining the implicit token from 0 to S. During the unscheduled part of an NUT, nodes 0 to U share the opportunity to transmit non-time-critical data in a round-robin fashion until the allocated unscheduled duration is expired.

When the guardband time is reached, all nodes stop transmitting, and only the node with the lowest MAC ID, called the "moderator," can transmit a maintenance message, called the "moderator frame," which accomplishes the synchronization of all timers inside each node and the publishing of critical link parameters such as NUT, node time, S, U, etc. If the moderator frame is not heard for two consecutive NUTs, the node with the lowest MAC ID will begin transmitting the moderator frame in the guardband of the third NUT. Moreover, if a moderator node notices that another node has a lower MAC ID than its own, it immediately cancels its moderator role.

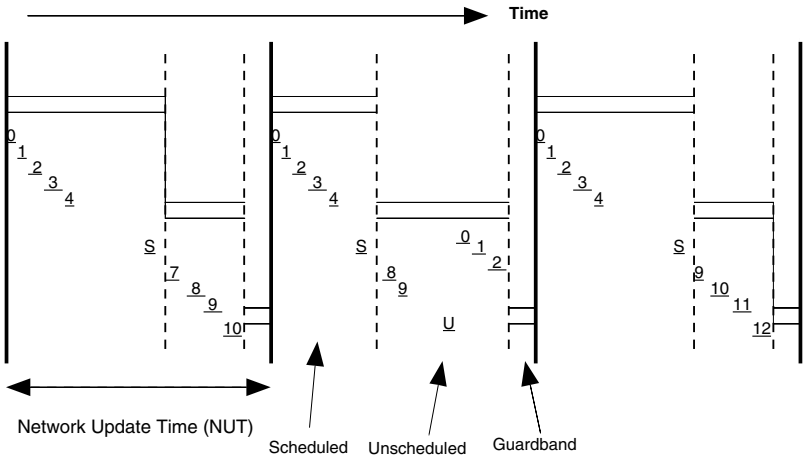


Fig. 3. Medium access during scheduled, unscheduled, and guardband time

Advantages: The token bus protocol is a deterministic protocol that provides excellent throughput and efficiency at high network loads [8,24]. During network operation, the token bus can dynamically add nodes to or remove nodes from the network. This contrasts with the token ring case, where the nodes physically form a ring and cannot be added or removed dynamically [24]. Scheduled and unscheduled segments in each NUT cycle make ControlNet suitable for both time-critical and non-time-critical messages.

Disadvantages: Although the token bus protocol is efficient and deterministic at high network loads, at low channel traffic its performance cannot match that of contention protocols. In general, when there are many nodes in one logical ring, a large percentage of the network time is used in passing the token between nodes when data traffic is light [8].

3 CAN-Based Networks: DeviceNet

CAN is a serial communication protocol developed mainly for applications in the automotive industry but also capable of offering good performance in other time-critical industrial applications. The CAN protocol is optimized for short messages and uses a CSMA/arbitration on message priority (AMP) medium access method. Thus, the protocol is message oriented, and each message has a specific priority that is used to arbitrate access to the bus in case of simultaneous transmission. The bit stream of a transmission is synchronized on the start bit, and the arbitration is performed on the following message identifier, in which a logic zero is dominant over a logic one. A node that wants to transmit a message waits until the bus is free and then starts to send the identifier of its message bit by bit. Conflicts for access to the bus are solved during transmission by an arbitration process at the bit level of the arbitration field, which is the initial part of each frame. Hence, if two devices want to send messages at the same time, they first continue to send the message frames and then listen to the network. If one of them receives a bit different from the one it sends out, it loses the right to continue to send its message, and the other wins the arbitration. With this method, an ongoing transmission is never corrupted.

In a CAN-based network, data are transmitted and received using *message frames* that carry data from a transmitting node to one or more receiving nodes. Transmitted data do not necessarily contain addresses of either the source or the destination of the message. Instead, each message is labeled by an identifier that is unique throughout the network. All other nodes on the network receive the message and accept or reject it, depending on the configuration of mask filters for the identifier. This mode of operation is known as multicast.

DeviceNet is an example of a technology based on the CAN specification that has received considerable acceptance in device-level manufacturing applications. The DeviceNet specification is based on the standard CAN (11-bit identifier only)[4] with an additional application and physical layer specification [4, 17].

The frame format of DeviceNet is shown in Fig. 4 [4]. The total overhead is 47 bits, which includes start of frame (SOF), arbitration (11-bit identifier), control, CRC, acknowledgment (ACK), end of frame (EOF), and intermission (INT) fields. The size of a data field is between 0 and 8 bytes. The DeviceNet protocol uses the arbitration field to provide source and destination addressing as well as message prioritization.

Advantages: CAN is a deterministic protocol optimized for short messages. The message priority is specified in the arbitration field. Higher priority messages always gain access to the medium during arbitration. Therefore, the transmission delay for higher priority messages can be guaranteed.

[4]The CAN protocol supports two message frame formats: standard CAN (version 2.0A, 11-bit identifier) and extended CAN (version 2.0B, 29-bit identifier).

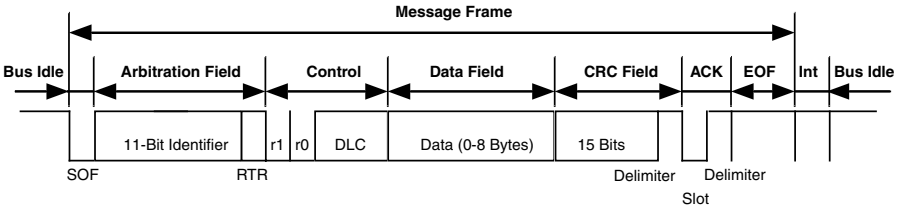


Fig. 4. The message frame format of DeviceNet (standard CAN format)

Disadvantages: The major disadvantage of CAN compared with the other networks is the slow data rate (maximum of 500 Kbps). Thus, the throughput is limited compared with other control networks. The bit synchronization requirement of the CAN protocol also limits the maximum length of a DeviceNet network. CAN is also not suitable for transmission of messages of large data sizes, although it does support fragmentation of data that is more than 8 bytes.

4 Timing Components

The important time delays that should be considered in an NCS analysis are the sensor-to-controller and controller-to-actuator end-to-end delays. In an NCS, message transmission delay can be broken into two parts: device delay and network delay. The device delay includes the time delays at the source and destination nodes. The time delay at the source node includes the preprocessing time, T_{pre}, and the waiting time, T_{wait}. The time delay at the destination node is only the postprocessing time, T_{post}. The network time delay includes the total transmission time of a message and the propagation delay of the network. The total time delay can be expressed by the following equation:

$$T_{delay} = T_{pre} + T_{wait} + T_{tx} + T_{post}. \tag{1}$$

The key components of each time delay are shown in Fig. 5 and will be discussed in the following subsections.

4.1 Pre- and postprocessing times at source and destination nodes

The preprocessing time at the source node is the time needed to acquire data from the external environment and encode it into the appropriate network data format. There may be one processor performing both functions, or multiple processors; we define the total elapsed time required as the pre- or postprocessing time. This time depends on the device software and hardware characteristics. In many cases, it may be assumed that the preprocessing time

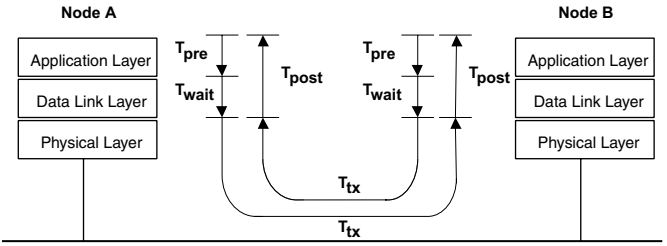


Fig. 5. A timing diagram showing time spent sending a message from a source node to a destination node

is constant or negligible. However, this assumption is not true in general; in fact, there may be noticeable differences in processing time characteristics between similar devices, and these delays may be significant.

The postprocessing time at the destination node is the time taken to decode the network data into the physical data format and output it to the external environment.

4.2 Experimental investigation of pre- and postprocessing times

In practical applications, it is very difficult to identify each individual timing component discussed above. Instead, by monitoring the time-stamped traffic of the request-response messaging on a DeviceNet network, we can show the characteristics of processing times, i.e., the sum of the preprocessing and postprocessing times of one device.

In the experimental setup, there is only one master and one slave connected to the network and the master continuously polls this slave. Referring to Fig. 5, let Node A be the master and Node B be the slave. Here, there is no other network traffic other than the request-response messages between the master and slave, i.e., $T_{wait} = 0$, and the request-response frequency is set low enough that no messages are queued up at the sender buffer. By monitoring the message traffic on the network medium and time-stamping each message, we can further calculate the processing time of each request-response, i.e., $T_{post} + T_{pre}$, after subtracting the transmission time.

Fig. 6 shows the histogram of 400 samples of four typical DeviceNet device processing times [11]. The devices are standard I/O types, such as those used for limit switches. The (right) solid and (left) dashed lines are the maximum and minimum values of the processing times, respectively. The histogram plots indicate the nondeterministic processing times of different network devices and their variance. Devices 1 and 3 have a similar functionality of discrete inputs/outputs, but different numbers of input/output modules. Device 3 provides several augmentable modules and hence has more processing units and a higher computation load. Device 1, on the other hand, has only one unit. Device 2 has a fairly consistent processing time, i.e., a low variance. Note that

the smallest time that can be recorded is 1 μs. The uniform distribution of processing time at Device 4 is due to the fact that it has an internal sampling time which is mismatched with the request frequency. Hence, the processing time recorded here is the sum of the actual processing time and the waiting time inside the device. Device 4 also provides more complex functionality and has a longer processing time than the others.

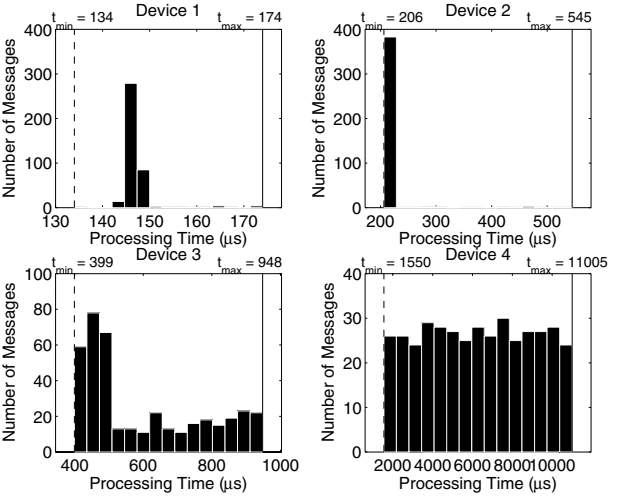


Fig. 6. Processing time histogram of four typical DeviceNet devices

A key point that can be taken from the data presented in Fig. 6 is that the device processing time can be substantial in the overall calculation of T_{delay}. In fact, this delay often dominates over network delays. Thus, in designing NCSs, device delay and delay variability should be considered as important factors when choosing components.

4.3 Transmission time on network channel

The transmission time is the most deterministic parameter in a network system because it only depends on the data rate, the message size, and the distance between two nodes. The formula for transmission time can be described as follows. $T_{tx} = T_{frame} + T_{prop}$. T_{frame} is the time required to send the packet across the network, and T_{prop} is the propagation time between any two devices. Since the typical transmission speed in a communication medium is 2×10^8 m/s, the propagation time T_{prop} is negligible on a small scale. In the worst case, the propagation delays from one end to the other of the network cable for these three control networks are $T_{prop} = 25.6$ μs for Ethernet (2500 m), $T_{prop} = 10$ μs for ControlNet (1000 m), and $T_{prop} = 1$ μs for DeviceNet (100 m). The length in parentheses represents the typical maximum

cable length used. The propagation delay is not easily characterized because the distance between the source and destination nodes is not constant among different transmissions. For comparison, we will assume that the propagation times of these three network types are the same, say, $T_{prop} = 1$ μs (100 m). Note that T_{prop} in DeviceNet is generally less than one bit time because DeviceNet is a bit-synchronized network. Hence, the maximum cable length is used to guarantee the bit synchronization among nodes.

The frame time, T_{frame}, depends on the size of the data, the overhead, any padding, and the bit time. Let N_{data} be the size of the data in terms of bytes, N_{ovhd} be the number of bytes used as overhead, N_{pad} be the number of bytes used to pad the remaining part of the frame to meet the minimum frame size requirement, and N_{stuff} be the number of bytes used in a stuffing mechanism (on some protocols). The frame time can then be expressed by the following equation:

$$T_{frame} = [N_{data} + N_{ovhd} + N_{pad} + N_{stuff}] \times 8 \times T_{bit}. \tag{2}$$

The values N_{data}, N_{ovhd}, N_{pad}, and N_{stuff}[5] can be explicitly described for the Ethernet, ControlNet, and DeviceNet protocols, see [10].

4.4 Waiting time at source nodes

A message may spend time waiting in the queue at the sender's buffer and could be blocked from transmitting by other messages on the network. Depending on the amount of data the source node must send and the traffic on the network, the waiting time may be significant. The main factors affecting waiting time are network protocol, message connection type, and network traffic load. For example, consider the strobe message connection in Fig. 7. If Slave 1 is sending a message, the other 8 devices must wait until the network medium is free. In a CAN-based DeviceNet network, it can be expected that Slave 9 will encounter the most waiting time because it has a lower priority on this priority-based network. However, in any network, there will be a non-trivial waiting time after a strobe, depending on the number of devices that will respond to the strobe.

The blocking time, which is the time a message must wait once a node is ready to send it, depends on the network protocol and is a major factor in the determinism and performance of a control network. It includes waiting time while other nodes are sending messages and the time needed to resend the message if a collision occurs.

[5]The bit-stuffing mechanism in DeviceNet is as follows: if more than 5 bits in a row are '1', then a '0' is added and vice versa. Ethernet and ControlNet use Manchester biphase encoding, and, therefore, do not require bit stuffing.

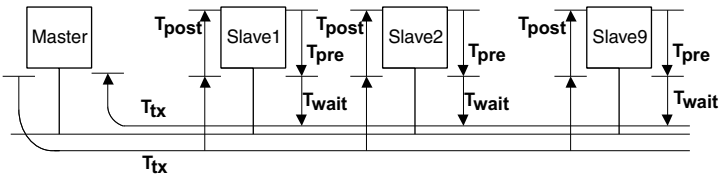


Fig. 7. Waiting time diagram

Ethernet blocking time

We first consider the blocking time for Ethernet, which includes time taken by collisions with other messages and the subsequent time waiting to be retransmitted. The BEB algorithm described in Section 2.1 indicates a probabilistic waiting time. An exact analysis of expected blocking time delay for Ethernet is very difficult [10]. At a high level, the expected blocking time can be described by the following equation:

$$E\{T_{block}\} = \sum_{k=1}^{16} E\{T_k\} + T_{resid}, \tag{3}$$

where T_{resid} denotes the residual time until the network is idle, and $E\{T_k\}$ is the expected time of the kth collision. $E\{T_k\}$ depends on the number of backlogged and unbacklogged nodes as well as the message arrival rate at each node. For the 16th collision, the node discards this message and reports an error message to the higher level processing units [21]. It can be seen that T_{block} is not deterministic and may be unbounded due to the discarding of messages.

ControlNet blocking time

In ControlNet, if a node wants to send a message, it must wait to receive the token from the logically previous node. Therefore, the blocking time, T_{block}, can be expressed by the transmission times and token rotation times of previous nodes. The general formula for T_{block} can be described by the following equation:

$$T_{block} = T_{resid} + \sum_{j \in \mathcal{N}_{noqueue}} T_{token}^{(j)} + \sum_{j \in \mathcal{N}_{queue}} \min(T_{tx}^{(j,n_j)}, T_{node}) + T_{guard}, \tag{4}$$

where T_{resid} is the residual time needed by the current node to finish transmitting, $\mathcal{N}_{noqueue}$ and $\mathcal{N}_{queue}$ denote the sets of nodes with messages and without messages in the queues, respectively, and T_{guard} is the time spent on the guardband period, as defined earlier. For example, if node 10 is waiting for the token, node 4 is holding the token and sending messages, and

nodes 6, 7, and 8 have messages in their queues, then $\mathcal{N}_{noqueue} = \{5, 9\}$ and $\mathcal{N}_{queue} = \{4, 6, 7, 8\}$. Let n_j denote the number of messages queued in the jth node and let T_{node} be the maximum possible time (i.e., token holding time) assigned to each node to fully utilize the network channel. For example, in ControlNet $T_{node} = 827.2$ μs, which is a function of the maximum data size, overhead frame size, and other network parameters. T_{token} is the token passing time, which depends on the time needed to transmit a token and the propagation time from node $i-1$ to node i. ControlNet uses an implicit token, and T_{token} is simply the sum of T_{frame} with zero data size and T_{prop}. If a new message is queued for sending at a node while that node is holding the token, then $T_{block} = T_{tx}^{(j,n_j)}$, where j is the node number. In the worst case, if there are N master nodes on the bus and each one has multiple messages to send, which means that each node uses the maximum token holding time, then $T_{block} = \sum_{i \in \mathcal{N}_{node} \setminus \{j\}} \min(T_{tx}^{(i,n_i)}, T_{node})$, where the min function is used because, even if it has more messages to send, a node cannot hold the token longer than T_{node} (i.e., $T_{tx}^{(j,n_j)} \leq T_{node}$). ControlNet is a deterministic network because the maximum time delay is bounded and can be characterized by (4). If the periods of each node and message are known, we can explicitly describe the sets $\mathcal{N}_{noqueue}$ and $\mathcal{N}_{queue}$ and n_j. Hence, T_{block} in (4) can be determined explicitly.

DeviceNet blocking time

The blocking time, T_{block}, in DeviceNet can be described by the following equation [22]:

$$T_{block}^{(k)} = T_{resid} + \sum_{\forall j \in \mathcal{N}_{hp}} \left\lceil \frac{T_{block}^{(k-1)} + T_{bit}}{T_{peri}^{(j)}} \right\rceil T_{tx}^{(j)}, \tag{5}$$

where k is the iteration index of obtaining steady-state T_{block}, T_{resid} is the residual time needed by the current node to finish transmitting, $\mathcal{N}_{hp}$ is the set of nodes with higher priority than the waiting node, $T_{peri}^{(j)}$ is the period of the jth node, and $\lceil x \rceil$ denotes the smallest integer number that is greater than or equal to x. The summation denotes the time needed to send all the higher priority messages. While a low priority node is waiting for the channel to become available, it is possible for other high priority nodes to be queued, in which case the low priority node loses the arbitration again. This situation accumulates the total blocking time. The worst-case T_{resid} under a low traffic load is

$$T_{resid} = \max_{\forall j \in \mathcal{N}_{node}} T_{tx}^{(j)}, \tag{6}$$

where $\mathcal{N}_{node}$ is the set of nodes on the network. However, because of the priority-arbitration mechanism, low priority message transmission may not be deterministic or bounded under high loading.

Fig. 8 shows experimental data of the waiting time of nine identical devices on a DeviceNet network. These devices have a very low variance of processing time. We collected 200 pairs of messages (request and response). Each symbol denotes the mean, and the distance between the upper and lower bars equals two standard deviations. If these bars are over the limit (maximum or minimum), then the value of the limit is used instead. It can be seen in Fig. 8 that the average waiting time is proportional to the node number (i.e., priority). Also, the first few devices have a larger variance than the others, because the variance of processing time occasionally allows a lower priority device to access the idle network before a higher priority one.

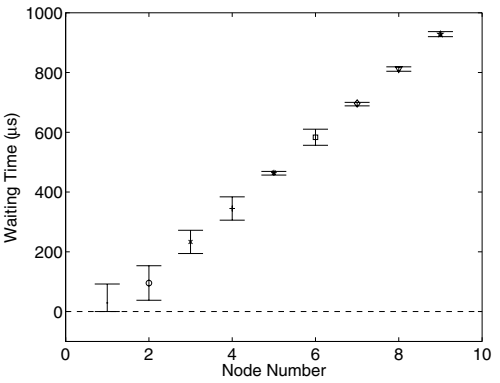


Fig. 8. Nine identical devices with strobed message connection

5 Network Comparisons

In this section, comparisons are drawn between the three types of control networks using the three networks that have been discussed in detail: Ethernet, ControlNet (token bus), and DeviceNet (CAN). The parameters for these networks are shown in Table 2. After summarizing the theoretical and simulation results for these three networks, we show some experimental results for time delays and throughput in wireless Ethernet.

5.1 Data transmission

One method for comparing control networks is by the time taken to transmit data and the efficiency of the data transmission.

As shown in Fig. 9(a), the transmission time for DeviceNet is longer than the others because of the lower data rate (500 Kbps). Ethernet requires less transmission time on larger data sizes (>20 bytes) compared with the others.

Table 2. Typical system parameters of control networks

	Ethernet	ControlNet	DeviceNet
Data rate[a] (Mbps)	10	5	0.5
Bit time (μs)	0.1	0.2	2
Max. length (m)	2500	1000	100
Max. data size (byte)	1500	504	8
Min. message size (byte)[b]	72[c]	7	47/8[d]
Max. number of nodes	>1000	99	64
Typical Tx speed (m/s)	coaxial cable: 2×10^8		

a: typical data rate; b: zero data size;
c: including the preamble and start of delimiter fields;
d: DeviceNet overhead is 47 bits.

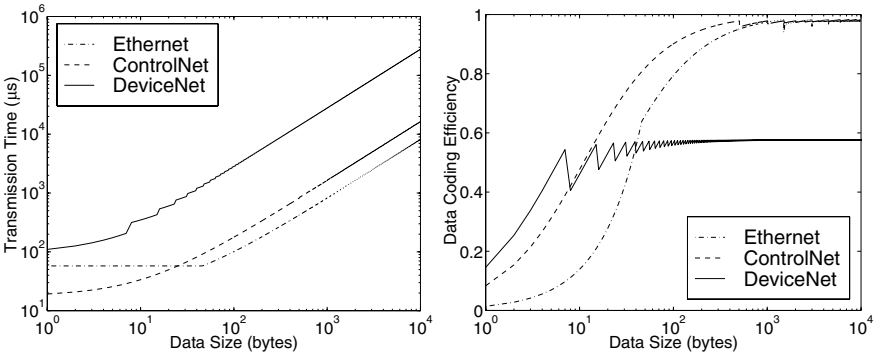
Although ControlNet uses less time to transmit the same amount of data, it needs some time (NUT) to gain access to the network.

The data coding efficiency (see Fig. 9(b)) is defined by the ratio of the data size and the message size (i.e., the total number of bytes used to transmit the data). For small data sizes, DeviceNet is the best among these three types and Ethernet is the worst (due to its large minimum message size). For large data sizes, ControlNet and Ethernet are better than DeviceNet (DeviceNet is only 58% efficient due to its small maximum message size, but ControlNet and Ethernet are near 98% efficient). For control systems, the data size is generally small. Therefore, the above analysis suggests that DeviceNet may be preferable in spite of the slow data rate. Before making that decision, however, the average and total time delay and the throughput of the network must be investigated.

5.2 Case study of 10-node NCS

In this section, we use a case study of an NCS to compare the three different control networks. The system has 10 nodes, each with 8 bytes of data to send every period. MATLAB[6] is used to simulate the MAC sublayer protocols of the three control networks. Network parameters such as the number of nodes, the message periods, and message sizes can be specified in the simulation model. In our study, these network parameters are constant. The simulation program records the time delay history of each message and calculates network performance statistics such as the average time delay seen by messages on the network, the efficiency and utilization of the network, and the number of messages that remain unsent at the end of the simulation run.

[6]MATLAB is a technical computing software developed by The MathWorks, Inc.



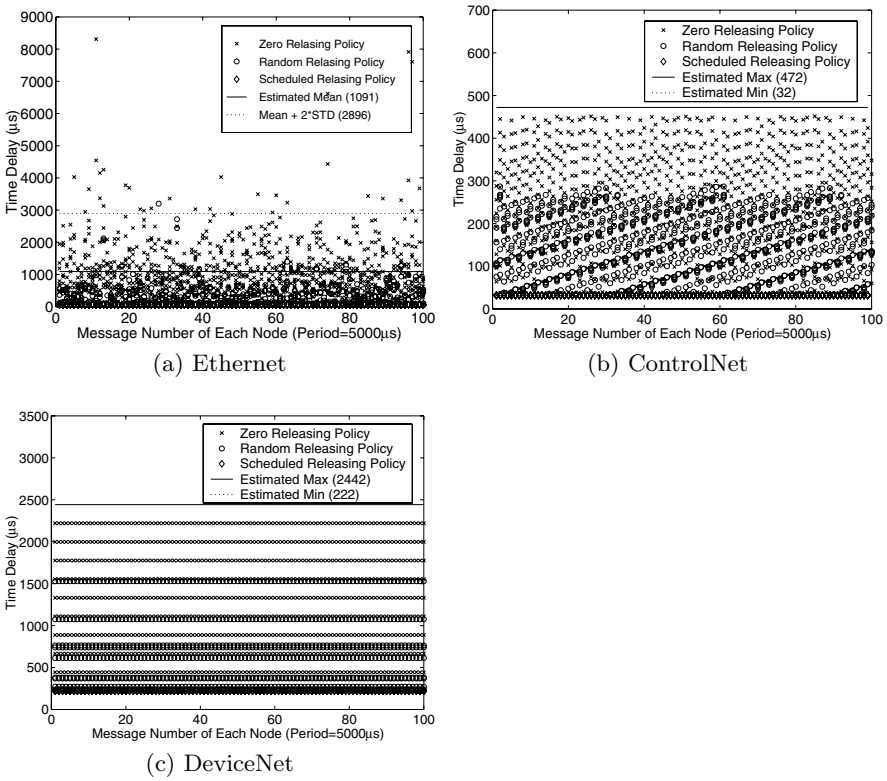
(a) Transmission time (b) Data coding efficiency

Fig. 9. A comparison of transmission time and data coding efficiency versus the data size for three control networks

Based on the three different types of message connections (poll, strobe, and cyclic), we consider the following three releasing policies. The first policy, which we call the "zero releasing policy," assumes that every node tries to send its first message at $t = 0$ and sends a new message every period. This type of situation occurs when a system powers up and there has been no prescheduling of messages or when there is a strobe request from the master. The second policy, the "random releasing policy," assumes a random start time for each node; each node still sends a new message every period. The possible situation for this releasing policy is the cyclic messaging, where no preschedule is done. In the third policy, called "scheduled releasing policy," the start-sending time is scheduled to occur (to the extent possible) when the network is available to the node; this occurs in a polled connection or with a well-scheduled cyclic policy.

In addition to varying the release policy, we can also change the period of each node to demonstrate the effect of traffic load on the network. For each releasing policy and period, we simulate the system and calculate the average time delays of these 10 nodes. We then compare the simulation results to the analytic results described in Section 4. For ControlNet and DeviceNet, the maximum time delay can be explicitly determined. For Ethernet, the expected value of the time delay can be computed using the BEB algorithm once the releasing policy is known.

The simulation results for a message period of 5000 μs are summarized in Fig. 10. The zero releasing policy has the longest average delay in every network because all nodes experience contention when trying to send messages. Although the Ethernet data rate is much faster than that of DeviceNet, the delays due to collisions and the large required message size combine to increase the average time delay for Ethernet in this case. For a typical random releasing policy, average time delays are reduced because not all nodes try to



(a) Ethernet

(b) ControlNet

(c) DeviceNet

Fig. 10. Message time delay associated with three releasing policies (10-node case). The estimated mean, maximum, and minimum values are computed from the network analysis for the zero and scheduled releasing policies.

send messages (or experience network contention) at the same time, although some contention still exists. The scheduled releasing policy makes the best use of each individual network; the time delay of this releasing policy is only the transmission time.

In Ethernet, shown in Fig. 10(a), the zero and random releasing policies demonstrate its nondeterministic time delay, even though the traffic load is not saturated. Fig. 10(b) shows that the message time delay of ControlNet is bounded for all releasing policies; we can estimate the lower and upper bounds based on the formulae derived in Section 4. Due to the asynchronicity between the message period and the token rotation period, these time delays exhibit a linear trend with respect to the message number. The simulation results for DeviceNet, shown in Fig. 10(c), demonstrate that every node in DeviceNet has a constant time delay which depends only on the node number. The estimated mean time delay (1091 μs) for Ethernet in Fig. 10(a) is computed for the case of the zero releasing policy from (3), and the variance is taken as twice the

standard deviation. The maximum and minimum time delays for ControlNet and DeviceNet are computed from (4) and (5).

5.3 Wireless Ethernet throughput and delays

In addition to time delays, the difference between the theoretical data rate and the practical throughput of a control network should be considered. For example, raw data rates for 802.11 wireless networks range from 11 to 54 Mbits/sec. The actual throughput of the network, however, is lower due to both the overhead associated with the interframe spaces, ACK, and other protocol support transmissions, and to the actual implementation of the network adapter. Although 802.11a and 802.11g have the same raw data rate, the throughput is lower for 802.11g because its backwards compatibility with 802.11b requires that the interframe spaces be as long as they would be on the 802.11b network. Computed and measured throughputs are shown in Table 3 [5]. The experiments were conducted by continually sending more traffic on the network until a further setpoint increase in traffic resulted in no additional throughput.

Table 3. Maximum throughputs for different 802.11 wireless Ethernet networks. All data rates and throughputs are in Mbit/sec.

Network type	802.11a	802.11g	802.11b
Nominal data rate	54	54	11
Theoretical throughput	26.46	17.28	6.49
Measured throughput	23.2	13.6	3.6

Experiments conducted to measure the time delays on wireless networks are summarized in Table 4 and Fig. 11 [5]. Data packets were sent from the client to the server and back again, with varying amounts of cross-traffic on the network. The send and receive times on both machines were time-stamped. The packet left the client at time t_a and arrived at the server at time t_b; then left the server at time t_c and arrived at the client at time t_d. The sum of the pre- and postprocessing times and the transmission time on the network for both messages can be computed as (assuming that the two nodes are identical)

$$\begin{aligned} 2 * T_{delay} &= 2 * (T_{pre} + T_{wait} + T_{tx} + T_{post}) \\ &= t_d - t_a - (t_c - t_b). \end{aligned}$$

Note that this measurement does not require that the clocks on the client and server be synchronized. Since the delays at the two nodes can be different, it is this sum of the two delays that is plotted in Fig. 11 and tabulated in Table 4.

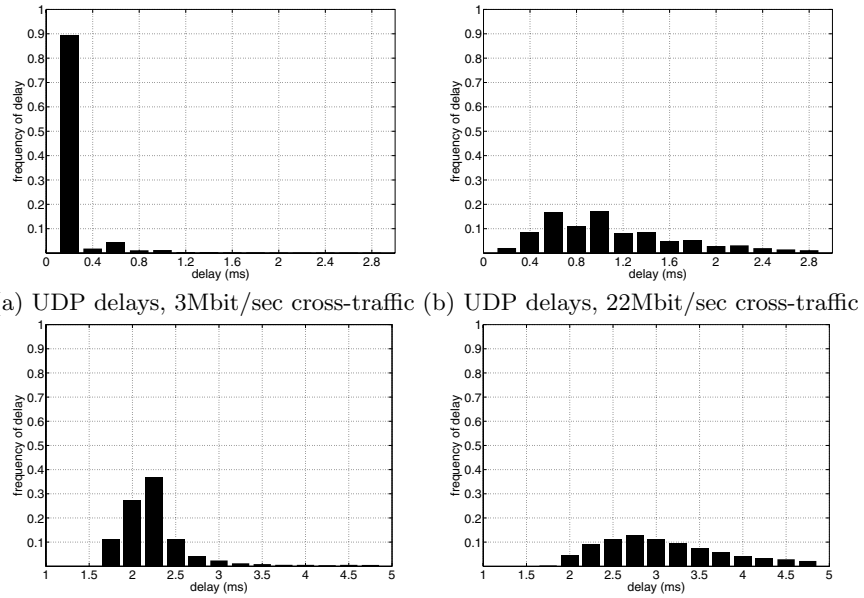
Two different types of data packets were considered: User Datagram Protocol (UDP), and object linking and embedding (OLE) for Process Control (OPC). UDP is a commonly used connectionless protocol that runs on top of Ethernet, often utilized for broadcasting. UDP packets carry only a data load of 50 bytes. OPC is an application-to-application communication protocol primarily utilized in manufacturing to communicate data values. OPC requires extra overhead to support this application layer; consequently, the OPC packets carry the maximum packet load of 512 data bytes. For comparison purposes, the frame times (including the overheads) are computed for the different packets.

Table 4. Computed frame times and experimentally measured delays on wireless networks; all times in ms.

Network type	802.11a	802.11g	802.11b
Frame time (UDP), computed	0.011	0.011	0.055
Median delay (UDP), measured	0.346	0.452	1.733
Frame time (OPC), computed	0.080	0.080	0.391
Median delay (OPC), measured	2.335	2.425	3.692

6 Conclusions and Future Work

The features of three candidate control networks — Ethernet (CSMA/CD), ControlNet (Token Bus), and DeviceNet (CAN) — were discussed in detail. With respect to Ethernet, which is becoming more and more prevalent in control network applications, we described and contrasted the three main implementation types: hub-based, switched, and wireless. For all protocols we first described the MAC mechanisms, which are responsible for satisfying both the time-critical/real-time response requirement over the network and the quality and reliability of communication between devices on the network. We then focused on exploring timing parameters related to end-to-end delivery of information over the networks. These timing parameters, which will ultimately influence control applications, are affected by the network data rate, the periods of messages, the data or message size of the information, and the communication protocol. For each protocol, we studied the key performance parameters of the corresponding network when used in a control situation, including the magnitude and characteristics of the expected and measured time delays. Simulation results were presented for several different scenarios. The timing analyses and comparisons of message time delay given in this chapter should be useful for designers of NCSs. For example, the basic differentiation of the network approaches will help the designer to match



(a) UDP delays, 3Mbit/sec cross-traffic (b) UDP delays, 22Mbit/sec cross-traffic

(c) OPC delays, 3Mbit/sec cross-traffic (d) OPC delays, 22Mbit/sec cross-traffic

Fig. 11. Distributions of packet delays for different values of cross-traffic throughput on a 802.11a network

basic requirement rankings against network approaches. Also, analyses such as device delay experiments reveal to the designer the importance of device delay and device delay variability in NCS design.

Control systems typically send small amounts of data periodically, but require guaranteed transmission and bounded time delay for the messages. The suitability of network protocols for use in control systems is greatly influenced by these two criteria. Although Ethernet (including hub-based and wireless) has seen widespread use in many data transmission applications and can support high data rates up to 1 Gbps, it may not be suitable as the communication medium for some control systems when compared with deterministic network systems. However, because of its high data rate, Ethernet can be used for aperiodic/non-time-critical and large data size communication, such as communication between workstations or machine cells. For machine-level communication with controllers, sensors, and actuators, deterministic networks are generally more suitable for meeting the characteristics and requirements of control systems. For control systems with short and/or prioritized messages, CAN-based protocols such as DeviceNet demonstrate better performance. The scheduled and unscheduled messaging capabilities in ControlNet make it suitable for time-critical and non-time-critical messages. ControlNet is also suitable for large data size message transmission.

Future NCS research efforts are expected to focus on controller design for NCSs, which can differ significantly from the design of traditional centralized control systems. For example, controller design optimized to the delay expected in an NCS is explored in [12], and balancing quality of service and quality of performance (QoP) in control networks can be effected using techniques such as deadbanding [15].

Another body of future NCS research will focus on the utilization of Ethernet for control [16], with a special emphasis on wireless Ethernet. While wireless Ethernet is beginning to proliferate in manufacturing diagnostics, its acceptance as an NCS enabler has been very slow to occur due to issues of reliability, performance, and security [13]. However, the enormous flexibility, cost savings, and reliability benefits that could potentially be achieved with wireless systems will continue to drive wireless NCS research, with a focus not only on control system design, but also on higher performing, more reliable, and more secure networks for control. Its is easily conceivable that, within 10 years, wireless will be the preferred medium for NCSs.

Acknowledgement. Many of the results described in this chapter are described in more detail in [10] and [11]. The authors would like to thank Alexander Duschau-Wicke, a student at the University of Kaiserslautern in Germany, who visited the University of Michigan in 2004 and performed the wireless Ethernet experiments described in Section 5.3.

References

1. D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Englewood Cliffs, NJ, second edition, 1992.
2. B. J. Casey. Implementing Ethernet in the industrial environment. In *Proceedings of IEEE Industry Applications Society Annual Meeting*, volume 2, pages 1469–1477, Seattle, WA, October 1990.
3. ControlNet specifications, 1998.
4. DeviceNet specifications, 1997.
5. A. Duschau-Wicke. Wireless monitoring and integration of control networks using OPC. Technical report, NSF Engineering Research Center for Reconfigurable Manufacturing Systems, University of Michigan, 2004. Studienarbeit report for Technische Universit at Kaiserslautern.
6. J. Eidson and W. Cole. Ethernet rules closed-loop system. *InTech*, pages 39–42, June 1998.
7. Y. Koren, Z. J. Pasek, A. G. Ulsoy, and U. Benchetrit. Real-time open control architectures and system performance. *CIRP Annals—Manufacturing Technology*, 45(1):377–380, 1996.
8. S. A. Koubias and G. D. Papadopoulos. Modern fieldbus communication architectures for real-time industrial applications. *Computers in Industry*, 26:243–252, August 1995.
9. K. C. Lee and S. Lee. Performance evaluation of switched Ethernet for networked control systems. In *Proceedings of IEEE Conference of the Industrial Electronics Society*, volume 4, pages 3170–3175, November 2002.

10. F.-L. Lian, J. R. Moyne, and D. M. Tilbury. Performance evaluation of control networks: Ethernet, ControlNet, and DeviceNet. *IEEE Control Systems Magazine*, 21(1):66–83, February 2001.
11. F.-L. Lian, J. R. Moyne, and D. M. Tilbury. Network design consideration for distributed control systems. *IEEE Transactions on Control Systems Technology*, 10(2):297–307, March 2002.
12. F.-L. Lian, J. R. Moyne, and D. M. Tilbury. Time-delay modeling and optimal controller design for networked control systems. *International Journal of Control*, 76(6):591–606, April 2003.
13. J. Moyne, J. Korsakas, and D. M. Tilbury. Reconfigurable factory testbed (RFT): A distributed testbed for reconfigurable manufacturing systems. In *Proceedings of the Japan-USA Symposium on Flexible Automation*, Denver, CO, July 2004.
14. J. Moyne, N. Najafi, D. Judd, and A. Stock. Analysis of sensor/actuator bus interoperability standard alternatives for semiconductor manufacturing. In *Sensors Expo Conference Proceedings*, Cleveland, OH, September 1994.
15. P. G. Otanez, J. R. Moyne, and D. M. Tilbury. Using deadbands to reduce communication in networked control systems. In *Proceedings of the American Control Conference*, pages 3015–3020, Anchorage, AK, May 2002.
16. P. G. Otanez, J. T. Parrott, J. R. Moyne, and D. M. Tilbury. The implications of Ethernet as a control network. In *Proceedings of the Global Powertrain Congress*, Ann Arbor, MI, September 2002.
17. G. Paula. Building a better fieldbus. *Mechanical Engineering*, pages 90–92, June 1997.
18. J. Pinto. Fieldbus—conflicting "standards" emerge, but interoperability is still elusive. *Design Engineering, UK*, October 1999. Available at http://www.jimpinto.com/writings/fieldbus99.html.
19. R. S. Raji. Smart networks for control. *IEEE Spectrum*, 31(6):49–55, June 1994.
20. K. K. Ramakrishnan and H. Yang. The Ethernet capture effect: Analysis and solution. In *Proceedings of the 19th Conference on Local Computer Networks*, pages 228–240, Minneapolis, MN, October 1994.
21. A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Upper Saddle River, NJ, third edition, 1996.
22. K. Tindell, A. Burns, and A. J. Wellings. Calculating controller area network (CAN) message response times. *Control Engineering Practice*, 3(8):1163–1169, August 1995.
23. E. Vonnahme, S. Ruping, and U. Ruckert. Measurements in switched Ethernet networks used for automation systems. In *Proceedings of IEEE International Workshop on Factory Communication Systems*, pages 231–238, September 2000.
24. J. D. Wheelis. Process control communications: Token bus, CSMA/CD, or token ring? *ISA Transactions*, 32(2):193–198, July 1993.

Control Using Feedback over Wireless Ethernet and Bluetooth

A. Suri[1], J. Baillieul[2*], and D. V. Raghunathan[1]

[1] The MathWorks, Inc.
3 Apple Hill Drive
Natick, MA 01760, U.S.A.
{atul.suri,dhananjay.raghunathan}@mathworks.com

[2] College of Engineering
Boston University
Boston, MA 02215, U.S.A.
johnb@bu.edu

1 Introduction

This chapter is concerned with the general topic of control systems in which feedback information is transmitted over wireless communication channels as well as with some specific problems, challenges, and opportunities associated with implementing feedback control using Bluetooth and other current wireless networking technologies.

Networked control systems in which sensors, actuators, and other system components are interconnected using advanced wireless communications technologies (IEEE 802.11, IEEE 802.15.4, Bluetooth, etc.) present a number of unique advantages (e.g., the elimination of wiring, the possibility of self-organization of networks of heterogeneous devices, and the possibility of operation while all of the components are in motion relative to one another) as well as special challenges (rapid and unpredictable changes in communication channels, possible network congestion, packet loss, limited battery capacity, etc.). This chapter discusses the design and operation of networks of controlled devices using various networking technologies. While each of the approaches has its own advantages, our main focus will be on Bluetooth because of the support it provides for interoperability of heterogeneous devices. The primary focus of our discussion will be on the effectiveness of wireless networks in providing communications services in applications with hard real-time constraints. We have in mind systems which will fail to operate properly—perhaps

*This work was supported by ODDR&E MURI01 Program Grant Number DAAD19-01-1-0465 to Boston University and the Center for Networked Communicating Control Systems and by the National Science Foundation ITR Program Grant Number DMI-0330171.

in a catastrophic way—if data rates in network communications channels fall below certain critical thresholds. The issues raised by constraints on network data rates are illustrated by experiments performed with an inverted pendulum where the feedback loop is closed over a Bluetooth radio link.

To place the discussion of control using Bluetooth in context, we briefly describe a number of other wireless networks. Each of the technologies listed in Table 1 has its own advocates and target applications. Among a number of current efforts to design and deploy networks of very low-power sensor nodes, several groups have independently developed network protocols for mesh topologies. In mesh-connected networks, each node is typically connected to more than two other nodes, and the protocols typically involve ad hoc routing and multihop message delivery. Such networks tend to be fault-tolerant and support rapid message rerouting so that the network is relatively robust with respect to component failures. Also, these networks are, in principle, completely scalable.

Mesh-connected wireless networks typically configure themselves by having nodes establish links to other nodes based on measured signal power levels. If the devices in the network are mobile, the signal power between nodes will vary as the relative node positions change, and the connection pattern (Fig. 1) will need to be updated accordingly. The overhead of continually reconfiguring the network has limited the use of mesh connection protocols for mobile ad hoc networks. Nevertheless, there is fairly widespread interest in networks whose stationary nodes have a mesh connection topology while the mobile nodes selectively connect to fixed nodes based on criteria such as signal strength, measured distance, etc. For a list of companies among which several are developing mesh-connected sensor networks, see `http://www.bu.edu/systems/consortium/participants.html`.

Many of the developers of low-power RF networking technologies have decided to adopt the IEEE 802.15.4 standard, and a number of companies have formed a consortium called the *ZigBee Alliance* to promote wirelessly networked monitoring and control products based on the IEEE 802.15.4 standard.[3] Released in 2003, this standard provides the basis for networking devices which communicate at low data-rates using extremely low power. Details may be found at `http://www.ieee802.org/15/pub/TG4.html`.

Bluetooth^{TM} is a wireless networking technology that was introduced by a consortium of companies made up of Intel, Nokia, Ericsson, Toshiba, and IBM. Founded in the Spring of 1999, the consortium conceived of Bluetooth technology to provide wireless communication among any group of spatially proximate Bluetooth-enabled devices. While portable digital phones and computers were early candidate applications, Bluetooth was designed to be an open-source technology, so that, in principle, it could be tailored to support interoperability of a wide variety of devices. Of course, to ensure such in-

[3]The principal (promoter) companies in the ZigBee Alliance are Ember, Freescale, Honeywell, Invensys, Mitsubishi, Motorola, Philips, and Samsung.

Table 1. Current wireless technologies supporting device networks

	Bluetooth	**IEEE 802.11**	**IEEE 802.15.4**	**Mesh***
Topology	Overlapping piconets	Cluster-tree	Unspecified	Mesh
Power consumption	Modest: $\sim 35mA$	Moderate $\sim 200mA$	Very low due to sleep mode	Very low due to sleep mode
Communication range	10–100 m	$>$ 100 m	10–100 m	10–100 m
Network size	8	32	255/65,000†	255/65,000†
Network config. time	3–60 seconds	Fast (<1 sec.)	Several seconds	Several seconds
Physical layer transmit freq.	2.4 GHz	2.4 GHz	886/915 MHz and 2.4 GHz	886/915 MHz and 2.4 GHz
Max data rate (appl. layer)	723 kilobits/s‡	>11,000 kilobits/s	250 kilobits/s	∼250 kilobits/s
Security	Freq. hopping, plus 128bit private key	SSL, and WEP 40/128 bit security	TBD	Freq. hopping, etc.

*Many of the best-known mesh-type RF networks are being designed to implement the IEEE 802.15.4 standard. The networking products of companies such as Millennial Net (`http://www.millennialnet.com`), Ember (`http://www.ember.com`), and Sensicast (`http://www.sensicast.com`), however, predate the release of the standard.

†There are two modes in these specifications—one having up to 255 nodes and the other having up to 65,000.

‡The Bluetooth standards organization is currently discussing a next generation (Bluetooth 2) in which the maximum Bluetooth data rates will be 2-Mbit/s and 10-Mbit/s variants of the current Bluetooth spec.

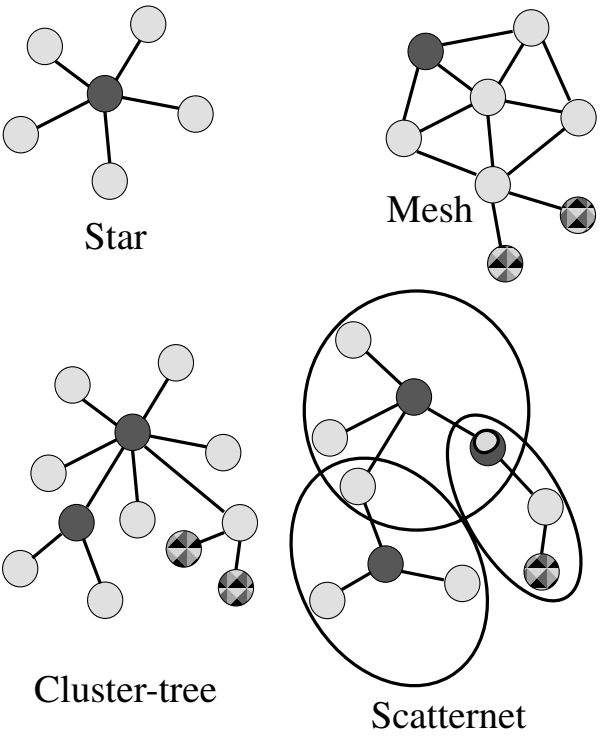


Fig. 1. Various types of network topologies. The shading key is *dark* = master/coordinator/router node, *light* = slave, full function node, and *textured* = slave, reduced function node. Notice that the scatternet (Bluetooth) topology is comprised of three separate piconets. Each piconet is organized around a separate master node, and the node shaded light inside dark in the right-hand piconet serves as the master in the right-hand piconet and a slave in the top piconet. Bluetooth standards prohibit nodes from being masters in more than one piconet.

teroperability, developers have been required to respect a fairly strict set of communication protocols. (See Sections 4 and 5.)

Referring to Table 1, the technical specifications of Bluetooth are comparable in many respects to other current wireless technologies. Bluetooth networks are formed whenever two or more devices are within range of one another. The devices will configure themselves into small networks called *piconets*, in which one device will always become a piconet master, with other devices operating as slaves. To explain the roles of slaves and masters, it is helpful to recall that Bluetooth communication is in the unregulated ISM band ($\sim$2.4GHz)[4] in which a standard transmission technique is spread-spectrum frequency hopping. It is the master's role to organize the frequency hopping

[4]The initials ISM stand for *Industrial, Scientific, and Medical*. The ISM bands are defined by the ITU-T in S5.138 and S5.150 of the Radio Regulations established

among all the nodes in the piconet. When the Bluetooth devices first communicate with one another and negotiate a piconet configuration, the device that is designated as master communicates its clock reading and device address to the remaining devices (which are all designated as slaves). This address is used to calculate the frequency hopping sequence of each of the slaves. Based on its local clock readings (relative to the master clock), each slave determines a sequence of frequencies on which to transmit or receive data in each clock interval.

Bluetooth specifications dictate that all piconets must have eight or fewer members (including the master node), and that in each piconet the master node will be unique. Piconets can have more than eight devices if the excess devices are attached in "parked" mode. If more than eight devices are operating, they may configure themselves into several piconets, and as indicated in Fig. 1, may integrate themselves into what is called a *scatternet*. Note that no device may be a master of more than one piconet, but there are two ways in which a device may be a member of more than one piconet. Specifically, any slave node in a piconet may simultaneously function as either a master or slave in another piconet. In cases of dual membership, the device must switch between two frequency hopping sequences. Because the piconets in a scatternet are not well synchronized, scatternets do not use Bluetooth bandwidth very efficiently. More research is needed to understand the real-time control of devices where feedback links are established over scatternets. Another potential problem with large-scale interconnected control system networks using Bluetooth is that competition for use of the 2.4GHz frequency band may result in a large number of lost packets. While this may be acceptable in an averaged sense, there can be periods of heavy congestion which render the quality of feedback control questionable. The reader is referred to [5] for a more detailed discussion of minimum-average and minimum-peak data rates in feedback control. For more detail on Bluetooth networking, the reader is referred to [2]. Fig. 2 shows examples of various packaging and types of interface hardware for Bluetooth transceivers.

1.1 Bluetooth as a networking technology for devices with hard real-time constraints

As indicated in Table 1, Bluetooth communications are subject to data rate limitations. Nevertheless, as described below, the achievable data rates can be

---

by the International Telecommunication Union. By international agreement, communications in the three frequencies of the ISM band: 860MHz/900MHz (33.3cm wavelength), 2.4GHz (12.2cm wavelength), and 5.8GHz (5.2cm wavelength) do not require a license anywhere in the world. Note that the 5.8GHz band is occasionally referred to as the UNII (*Unlicensed National Information Infrastructure*) band. There are other unregulated radio bands, but they are not of interest in discussing wireless Ethernet and Bluetooth.

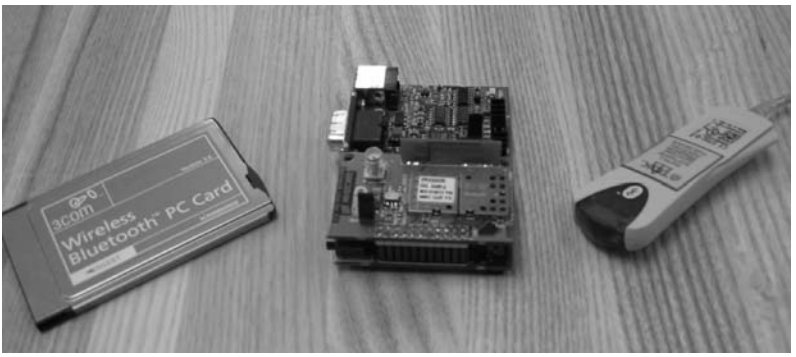


Fig. 2. Bluetooth radio transceivers can be connected to other devices in a variety of ways—including the PCMCIA card, nine-pin serial connector, and USB plug for the devices displayed in this photo.

high enough for some fairly demanding real-time applications. In the application described in Sections 4 and 5, Bluetooth radios were linked with a plant microprocessor and a controller so that both the forward and feedback links in the control loop made use of the wireless channel. As will be discussed in Section 5, the system was carefully engineered so that the Bluetooth packets made optimal use of the available channel capacity.

The emphasis Bluetooth technology places on interoperability of heterogeneous devices makes it attractive for integrating components in an ad hoc control network (ACN) provided that repeated network reconfiguration is not required. (See remarks below in Section 2.3 regarding issues with real-time reconfiguration of Bluetooth networks.) Given current levels of interest in protocols like the emerging IEEE 1451.2 standard for connecting transducers to network-capable applications processors (NCAPs), it should be possible in the near future to implement ACNs without any detailed knowledge of the underlying networking technologies. The use of such low-level connection standards will empower a future generation of "internetworked" devices which can be integrated into fairly complex systems without the user needing to specify any details regarding the way in which communications are handled. Such future device networks will involve families of smart transducer interface modules (STIMs) which will automatically configure themselves by first advertising their services and capabilities (sensing, actuation, computation, etc.) and then negotiating their role in the network in terms of required bandwidth, physical variables needing calibration, device-specific parameter units (temperature, pressure, etc.) and so forth. (See [4] for more information on Bluetooth-enabled STIMs.)

In principle, scatternet topologies of Bluetooth-enabled STIMs are completely scalable, but because there is no coordination of frequency hopping between component piconets in a scatternet, interference leading to packet

losses can be expected to occur as the numbers of networked devices increases within any given local area.

2 Control using IEEE 802.11

Research in the field of networked control systems has largely been focused on dedicated networks such as the Controller Area Network (CAN) bus and Token Bus. Data networks such as Ethernet (IEEE 802.3) have also been studied for distributed control over networks. In this section we describe the IEEE 802.11 standard, a.k.a *wireless Ethernet*, and the issues that arise when using it for soft real-time or supervisory control.

2.1 The IEEE 802.11 standard

802.11 is a part of the IEEE 802 family, which is a series of specifications for local and metropolitan area networks (LAN/MAN). The relationship between the IEEE 802.11 standard and other members of the family along with its place in the ISO Open Systems Interconnect (OSI) model is shown in Fig. 3. IEEE 802 specifications are focused on the two lowest layers of the OSI model

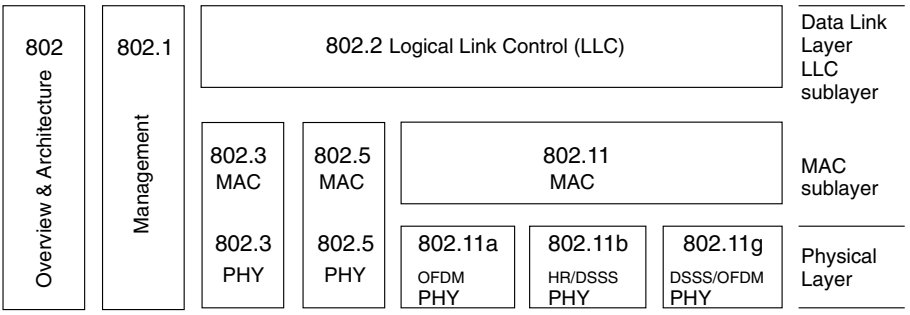


Fig. 3. The IEEE 802 family and its relation to the OSI model

because they incorporate both physical and data link components. The media access control (MAC) sublayer of the data link layer specifies a set of rules to determine how to access the medium and send data, but the details of the transmission and reception are left to the physical layer (PHY).

Individual specifications in the 802 series are identified by a second number. For example, 802.3 is the specification for a carrier sense multiple access network with collision detection (CSMA/CD) which is often called Ethernet. 802.2 specifies a common link layer (which is a part of the data link sublayer in the OSI model), the logical link control (LLC), which can be used by any lower-layer LAN technology. Management features for 802 networks are specified in 802.1.

802.11 is just another link layer that uses the 802.2/LLC encapsulation. The base 802.11 specification includes the 802.11 MAC and the two physical layers: a frequency hopping spread-spectrum (FHSS) physical layer and a direct-sequence spread-spectrum (DSSS) link layer. Later revisions to 802.11 added additional physical layers. 802.11b specifies a high-rate direct-sequence spread-spectrum (HR/DSSS). 802.11a describes a physical layer based on orthogonal frequency division multiplexing (OFDM) and 802.11g (which was ratified in June 2003) uses both HR/DSSS and OFDM for transmission. 802.11a operates in the 5GHz band at raw speeds of 54Mbits/s. 802.11b and 802.11g both operate in the 2.4GHz ISM band at speeds up to 11Mbits/s and 54 Mbits/s, respectively, and 802.11g is fully backward compatible with 802.11b.

The basic building block of an 802.11 network is the *basic service set* (BSS), which is simply a group of stations that communicate with each other. Communications take place within what is called the *basic service area*. When a station is in the basic service area, it can communicate with the other members of the BSS. A BSS is identified using a sequence of alphanumeric characters called the *basic service set identifier* (BSSID). All the stations in a given BSS have the same BSSID. BSSs come in two flavors, both of which are discussed below and shown in Fig. 4.

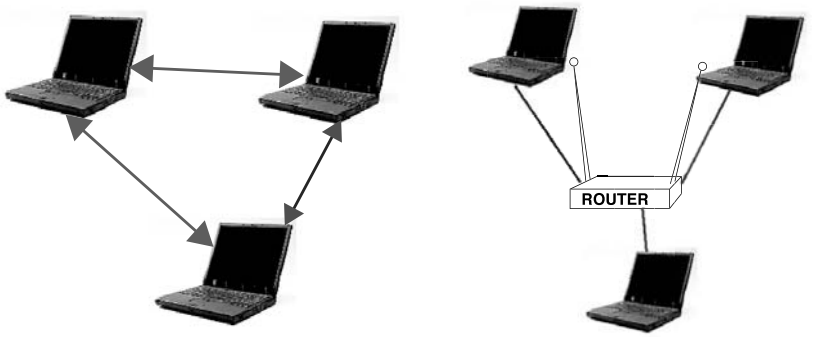


Fig. 4. (a) Independent BSS (*ad hoc mode*) and (b) Infrastructure BSS

Independent networks

Independent networks constitute an *independent* BSS (IBSS). Stations in an IBSS communicate directly with each other and thus must be within direct communication range. The smallest possible 802.11 network is an IBSS with

two stations. Typically, IBSSs are composed of a small number of stations set up for a specific purpose and for a short period of time. Due to their short durations, small size, and focused purpose, IBSSs are sometimes referred to as ad hoc BSSs or *ad hoc networks*, and consequently the stations are said to be operating in an ad hoc mode.

Infrastructure networks

Infrastructure networks are distinguished by the use of an access point. Access points are used for all communications in infrastructure networks, including communication between adjacent stations within direct communicating distance. Hence, any communication in the *infrastructure mode* requires at least two hops; one from the transmitting station to the access point and the other from the access point to the receiving station. The major advantage of infrastructure mode is that all stations are required to be within range of the access point and not necessarily within direct communication range of each other.

2.2 Supervisory control over IEEE 802.11b networks

As mentioned above, IEEE 802.11b supports both ad hoc (peer-to-peer) and infrastructure wireless networks. Communications over these networks can be performed via either of the two protocols supported by the TCP/IP suite: transmission control protocol (TCP) or user datagram protocol (UDP). TCP is a connection-based protocol that guarantees data delivery, whereas UDP is a connectionless protocol where there are no guarantees of data delivery. In TCP, a mutually acknowledged connection between the sender and the receiver is established during handshaking, which is done before data transmission. The sender retransmits the data unless it receives an acknowledgment from the receiver verifying successful data delivery. UDP, however, has no provisions for retransmission; if data is not delivered, retransmissions have to be done at a higher layer. The combination of low overhead and discarding data rather than retransmitting it allows more frequent communication of small data packets, making UDP a preferred choice for real-time network-mediated control. For supervisory control, on the other hand, the data packets are generally larger and infrequent; hence, TCP is often a better choice.

2.3 An experimental testbed for supervisory control

The testbed for supervisory control of mobile robot formations in the Intelligent Mechatronics Laboratory at Boston University consists of five mobile robots equipped with on-board computers and a suite of imaging and proximity sensing equipment. Each is also equipped with IEEE 802.11b PCMCIA cards and with an HP iPAQ h5550 Pocket PC with embedded IEEE 802.11b

and Bluetooth connected through an RS232 port. Another HP iPAQ h5550 Pocket PC with 802.11b and Bluetooth which served as the user interface was referred to as the *supervisory control commander*. (See Fig. 5.) Formation control algorithms were implemented on the on-board computer of the robots for maintaining controlled geometric formations of the robots. The supervisory control commander was used by the human operator to send formation reconfiguration commands to the corresponding Pocket PCs on the robots which in turn sent the command to the formation controller (running on the on-board computer) over a serial port. In different experiments, the user- and robot-Pocket PCs communicated with each other over either IEEE 802.11b or Bluetooth. Both the ad hoc and *infrastructure* modes of 802.11b were used.

Fig. 5. Formation controller interface. A number of user interface screens like the one shown here have been created. The boxes labeled C1 through C5 allow the user to specify which robot (s)he wishes to communicate with.

Each control packet consisted of 1 byte of supervisory control data and 34 bytes of overhead. The system was soft real-time—meaning that formation reconfiguration had to be achieved within only an approximate time of each motion command being issued. Because of the limited processing power of Pocket PCs, the overhead of retransmission at the application layer was much larger. Hence, TCP was preferred over UDP. Since Pocket PCs do not support the full features of the Windows sockets API, multithreading was required to send and receive data at the same time. Because the raw data speeds of IEEE

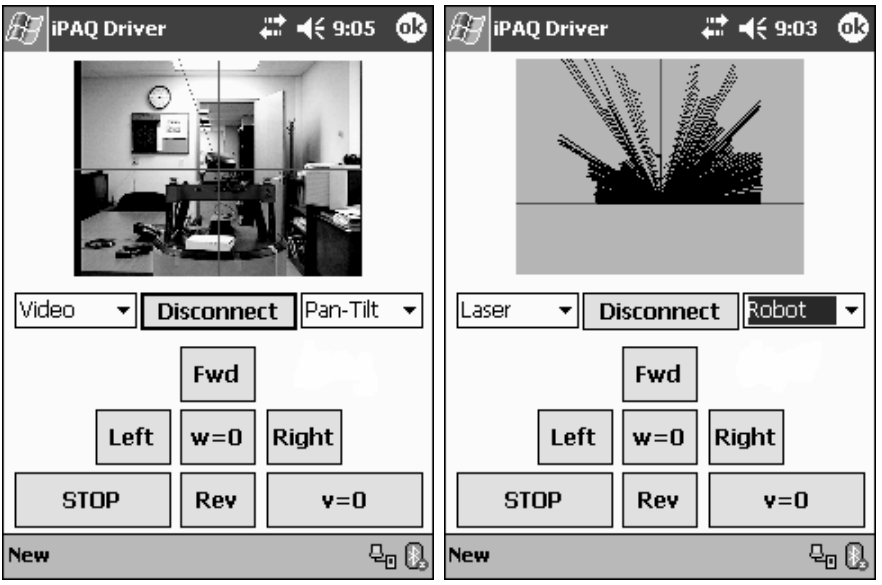
802.11b were much higher than the computation speeds of the Pocket PCs, most of the overhead was in data processing rather than in data transmission.

The Serial Port Profile of Bluetooth was used to communicate between the Pocket PCs over Bluetooth. There were several connectivity issues with Bluetooth. For example, if the Bluetooth devices went out of range, the devices had to be reset, and a new connection had to be established. On the other hand, 802.11b, in both infrastructure and ad hoc mode, was able to re-establish connection once in range. The round-trip delay for the same payload was also significantly higher for Bluetooth connections. However, it is important to note that the authors were using the Serial Port Profile of the Bluetooth application stack which is shipped with the HP iPAQs for Bluetooth links, whereas TCP connections were created for 802.11b connections. Hence, the authors had explicit control over packets in 802.11b connections. Bluetooth devices, however, had a larger range compared to 802.11b in the ad hoc mode. The 802.11b range in infrastructure mode easily superseded both Bluetooth and 802.11b ad hoc mode. Refer to Ploplys et al. [6] for signal-to-noise ratio (SNR) and data rate as a function of separation for 802.11b nodes.

As mentioned earlier, it is fairly straightforward to operate at the L2CAP or HCI layer of the Bluetooth protocol stack. Hence, the overhead of Bluetooth packets is generally small. On the other hand, it is extremely difficult to operate at a layer lower than TCP/IP in the OSI stack; hence, the overhead is fairly large for each packet. This is a significant advantage that Bluetooth has over 802.11b. 802.11b, in its ad hoc mode, requires that each node maintain a connection with every other node in the network. From our experiments, we found that this leads to a very unstable network once the number of nodes exceeds five. The Bluetooth protocol (which creates ad hoc networks) solves this problem by having a master in the piconet, and only the master is required to maintain connection with every node in the network.

The testbed also supported the teleoperation of mobile robots. In the teleoperation mode, the interfacing Pocket PC was removed from each robot, and the user Pocket PC communicated directly with the robot's on-board computer. Fig. 6 shows the teleoperation commander called iPAQDriver. The robot sent real-time image (Fig. 6(a)) and laser (Fig. 6(b)) data to the iPAQDriver. The user was able to send motion commands to both the robot and the camera head explicitly. A total of 57,600 bytes of payload data was transmitted over the network. The data was subdivided into packets of 1000 bytes payload so as not to exceed the fragmentation limit of 1500 bytes. Hence, 58 packets were sent for each data sample. For further details see [9].

The *fragmentation limit* is one of the configuration parameters of 802.11b networks. These networks have the ability to fragment packets to limit their length. When there is no interference on the network, fragmenting lowers the network throughput, because of the increased overhead of packet headers. However, in the presence of interference, fragmentation can actually increase the throughput. By decreasing the length of each packet, the probability of



(a) (b)

Fig. 6. (a) Video feedback and (b) laser range finder feedback for iPAQDriver

interference during an 802.11b packet transfer is reduced, and there is a trade-off between the lower packet error rate that can be achieved with shorter packets and the increased overhead of more headers on the network. Finding the optimal fragmentation setting to maximize the network throughput on a wireless network is an active area of research. (See, e.g., [7],[10].)

Ye et al. [11] proposed a modified IEEE 802.11b protocol called "prioritized CSMA/CA" for real-time control over 802.11b. Their network carried both real-time traffic and standard multimedia traffic. Ploplys et al. [6], on the other hand, used a standard dedicated 802.11b network for closed-loop control over wireless networks. In that work, the authors characterized the SNR of the nodes and hence the data rates of the network when the plant and controller separation varied. They also extended their controller to a multiple controller-plant system.

3 Controlling a Hard Real-Time System over Information Channels with Uncertainties

Consider a plant-controller system stabilized by closed-loop control over a communication channel. For our analysis, we shall view this as being comprised of two components. The first is the plant-controller system, which we

assume can be stabilized satisfactorily when the information channel is perfect. By "perfect" we mean that there is instant and reliable information transfer between the plant and the controller. The other component is the information channel or data network used to communicate information between the plant and controller. Our goal is to model and stabilize the overall system under uncertainties in the information channel. From the controller's perspective, the data network represents an uncertainty in the plant-controller system; from the perspective of the data network, the plant-controller system represents a load, or service requestor. We can examine both a controller-centric approach and a network-centric approach to this system, and these two perspectives form the core of our investigation. Next we report the results of laboratory experiments with a specific hard real-time system in which feedback was transmitted over network data channels.

3.1 The pendulum-cart system

The stabilization (in the inverted position) of a pendulum hinged on a cart through a force acting on the cart is a classical example of a feedback stabilization problem. Fig. 7 shows the setup of the system that was used to conduct our experiments. The system consists of a cart that moves on rails

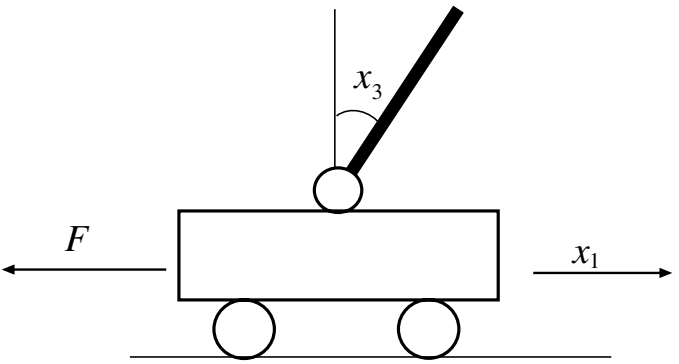


Fig. 7. The pendulum-on-a-cart system

and a pendulum hinged to the center of the cart. The cart is moved via a motor attached to one end of the rail. An optical encoder placed coaxially with the motor is used to determine the position of the cart on the rails. Another optical encoder placed coaxially with the axis of rotation of the pendulum determines the angle between the pendulum and the vertical. The position and velocity of the cart together with the angular position and velocity of the pendulum are sufficient to describe the dynamics of this system. The equations of motion are

$$\begin{aligned}
\frac{dx_1}{dt} &= x_2 \\
\frac{dx_2}{dt} &= \frac{2mLx_4^2 \sin x_3 + 4(F - cx_2) - 3mg \sin x_3 \cos x_3}{(4M+m) + 3m \sin^2 x_3} \\
\frac{dx_3}{dt} &= x_4 \\
\frac{dx_2}{dt} &= \frac{6(M+m)g \sin x_3 - 3mLx_4^2 \cos x_3 \sin x_3 - 6 \cos x_3 (F - cx_2)}{(4M+m) + 3m \sin^2 x_3},
\end{aligned} \tag{1}$$

where the force F is viewed as a control input, and the other parameters are give in Table 2. (See [3].)

Table 2. Parameters for the pendulum-cart system

Parameter	Symbol	Value
Mass of pendulum	m	0.23 kg
Mass of cart	M	2.4 kg
Length of pendulum	L	0.36 m
Gravity	g	9.81 m/s
Dissipation	c	0.05 Ns/m

We chose to design a linear feedback regulator for quadratic optimal cost (LQR) for this plant, and towards that end, the equations (1) were linearized about the unstable pendulum equilibrium, $(x_1, x_2, x_3, x_4) = (0, 0, 0, 0)$.

In order to effect the control, sensor information (from the optical encoders) was sent to the controller (a computer connected to the system) via a data channel. The controller computed the corresponding actuator input in order to stabilize the system and sent this information to the actuator via the same data channel. When this data channel is simply a dedicated ISA bus connected directly to the computer, there is very little delay for this information propagation and the overall effect on the stability of the system is minimal, if any. Indeed, with the control programs running under a real-time kernel (RTK), the pendulum-cart system can be sampled at frequencies up to 1000Hz. As described in the pendulum-cart system documentation, control programs can be developed and run using a *custom controller development facility* developed as a custom control DLL (dynamic linked library) written in C/C++. There are two features that must be taken into account in the development of the custom controller DLL

1. A sampling time T must be prescribed, and the custom controller needs to complete its execution in less than the period T, as it is called every T time units.
2. Sampling less frequently than 25Hz (40 ms inter-sample time) destabilizes the system from the inverted equilibrium position.

This setup served as the reference for all of our experiments.

If we replace the ISA bus with another information channel, say a LAN or a wireless medium (Bluetooth in our case), the control system dynamics

become more complex. The LAN is a shared channel, and so there is no dedicated resource to guarantee that control actions are completed within the minimum allotted sampling time T. The wireless medium, on the other hand, has limitations in terms of interference and data rates. These uncertainties bring dynamics into the system not seen when we have a dedicated bus servicing our network. In this case, it is important to capture the dynamics introduced by the data network.

3.2 Modeling and observing the data network

There are several ways in which one can characterize a data network and information channels depending on the application at hand. One can, for instance, use the signal-to-noise ratio (SNR) to capture the strength of a signal. In the context of real-time systems, the most important characteristic is the timely delivery of information and a deterministic bound on the maximum possible delay. The delay, bandwidth (data rate supported on the network), and the *jitter* (variability) in the delay are three properties that can be used to characterize the data network. The delay and communication bandwidth are related as shown in (2). The jitter can be characterized by, for example, the standard deviation in the delay values.

$$\begin{aligned} \textit{Total delay} &= \textit{Propagation delay} + \textit{Transmission delay} \\ \textit{Propagation delay} &= \frac{\textit{distance}}{\textit{speed of propagation in the medium}} \\ \textit{Transmission delay} &= \frac{\textit{packet size}}{\textit{bandwidth}}. \end{aligned} \tag{2}$$

For communication links with high bandwidth and small packets, the link latency is the essential component of the delay. (Recall that *latency* is the time required to encode, transmit, and process a message containing routing and other network-specific administrative data, but no message content. It is thus the time required to transmit a packet containing only the "overhead bits" depicted in Fig. 10.)

As far as the controller-plant system is concerned, the network may be modeled as a delay element with the length of the delay being variable. It now becomes important to be able to design and observe the delay over the network. We can affect the delay by structuring the transmitted data so as to minimize packet fragmentation. Then the delay can be measured in terms of the round-trip time (RTT)—the time it takes for a message traveling between nodes in the network to be delivered and for an acknowledgment or response to that message to come back to the sender. Notice that there are several potential sources of delay while one attempts to measure the RTT. The major component in our case came from the (network-induced) transmission delay, and hence delays like scheduling delays at the receiving end can be ignored. The RTT can be taken to represent the state of the network. The deviation in the RTT (measured, for example, by the standard deviation of the RTT observed over time) can be taken as a measure of the jitter. These two quantities

(the running average and deviation of the RTT) are adequate to represent the state of the network from the point of view of the control system.

Fig. 8 illustrates the importance of RTT in characterizing performance limits in networked control systems. In both cases depicted here, transmission delays dominate in determining bandwidth. Fig. 8(a) depicts a high-bandwidth link. The RTT is small, and in this case one can have a sampling period as long as 40ms. In Fig. 8(b), however, the largest tolerable sampling interval is around 25ms. Channel capacity limits the possible performance of the control system. For the high-bandwidth channel (Fig. 8(a)), the control system needs to reserve the channel for only 3 ms every 40 ms (the sampling period). In the case of Fig. 8(b), however, the control system keeps the communications channel in nearly constant use.

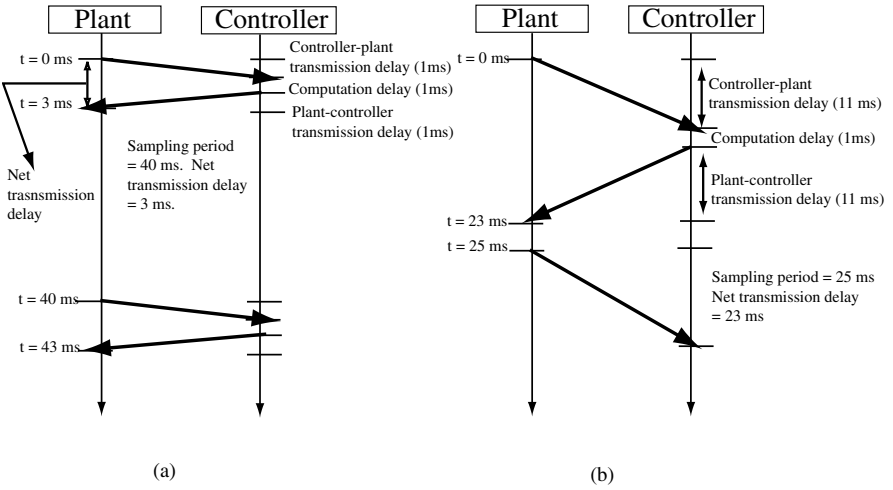


Fig. 8. (a) Timing diagram for a high-bandwidth data link between the plant and controller. (b) Timing diagram for a low-bandwidth data link between the plant and controller. In this case the transmission delay is almost equal to the sampling period.

4 Feedback over Bluetooth Wireless Data Channels

The layers comprising the Bluetooth stack are depicted in Fig. 9. From a high-level view, there are three main layers: the application protocols, the middleware protocols, and the transport protocols. The application layer of the stack is primarily comprised of higher-level protocols that are for the most part abstracted out from the actual physical medium being used to communicate information. These include protocols like the File Transfer Protocol

(FTP). There is a rich list of applications defined by the profiles in Bluetooth, especially in the context of ad hoc networking.

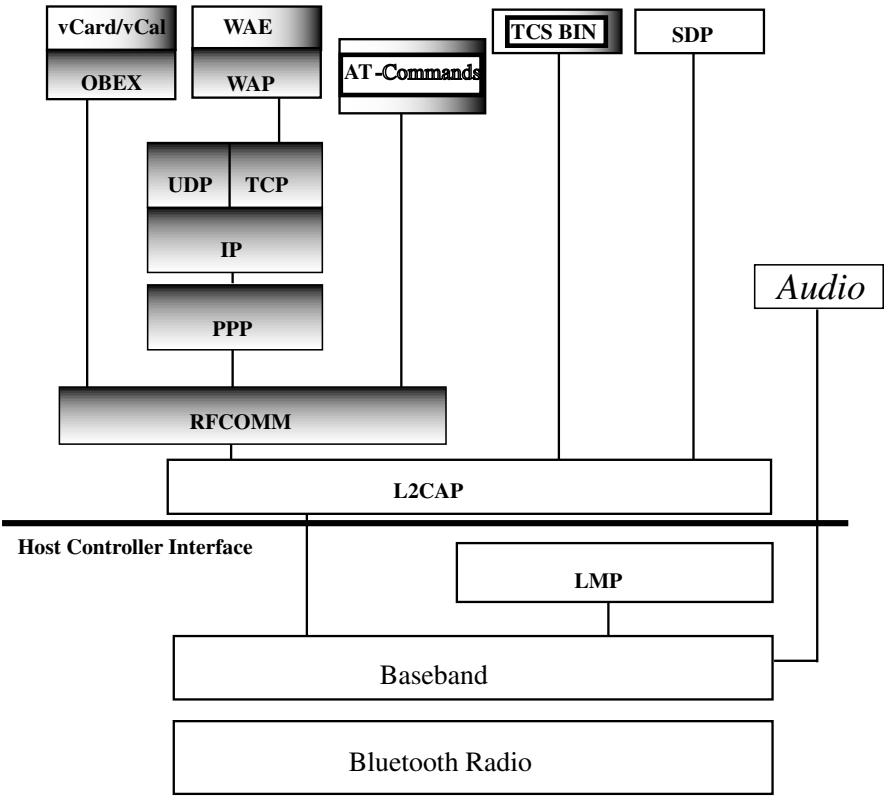


Fig. 9. The Bluetooth stack with protocols that appear in each of the main layers as described in the text

The middleware layer of the stack implements several of the Internet protocols (such as TCP, IP, PPP, IrDA), as well as a serial port emulator protocol RFCOMM to interface with the Bluetooth transport protocols. In the same layer, there is also a service discovery protocol (SDP) that lets devices query each other for the services that are available as well as the method of access of the service from each device.

The transport protocols allow Bluetooth devices to actually locate each other and to create physical and logical links to sustain the connections. The transport layer contains much of the novel potential that Bluetooth offers. There are two protocols that have been established for this layer: the Link Layer Channel Access Protocol (L2CAP) and the Host Controller Interface (HCI). The L2CAP supports asynchronous connections, in which the guarantee is on the packet delivery but not the time it takes for the delivery. The

HCI is required when a Bluetooth module is designed as an external device that is plugged into, say, a laptop or PDA (such as the Pocket PC described in Section 2.3). This interface exports the functionality of the Bluetooth module to the external device, which can, through the HCI, request services from the Bluetooth module.

From the perspective of the HCI layer, there are two possible kinds of links—the asynchronous connectionless link (ACL) and the synchronous connection oriented (SCO) link. The ACL guarantees data delivery across the network without guarantees on the time, while the SCO link works to time deadlines for data delivery. SCO links will drop packets if necessary in order to maintain the flow of data packets. This is a required feature for voice, for example, where it is better to lose a few packets on the way, as the alternative would mean delay, which can be perceived by the ear. It is on this layer that we have focused much of our investigation. The point of view here is that this channel may be used for hard real-time control, with the use of techniques like redundancy to provide robustness of the network to packet drops.

5 Packet Structures for Real-Time Control Using Bluetooth

The pendulum-cart experimental control system with Bluetooth feedback channels is depicted in Fig. 10. Before describing the joint operation of the various channels shown, it is useful to give a general overview of the pendulum-cart system operation when Bluetooth communication is used to carry state and actuation data between the controller and plant respectively. The control computer in the upper left of Fig. 10 (also referred to as the host controller) communicates with the Bluetooth EBSK module (Ericsson Bluetooth Starter Kit) via command packets, event packets, and data packets. Two types of packets can be sent from the host to the host controller (Bluetooth module): command packets and data packets, and two types of packets (event packets and data packets) are returned from the host controller to the host. The first byte (not shown in the figure) is used to indicate (to the Bluetooth firmware) which type of packet is being sent (command, event, or data packet). A command packet is not passed over the wireless interface; it is internal and affects only the host controller and link settings. The host controller, in turn, sends an event packet back to the host to indicate the status of the command and the possible changes made in the configuration due to the command. An HCI data packet, however, is read, and its data payload is repackaged into a 366-bit DM1 air packet, which is sent over the wireless medium. Note that as depicted in Fig. 10 the data packet has a theoretical payload size limit of 65,535 bytes. However, the size of the payload in the HCI data packet cannot exceed 17 bytes, if one is to avoid fragmentation in the air packets which are passed over the wireless channel. Such fragmentation would require two or more air

packets to be sent over the wireless link. The resulting fragmentation overhead can seriously increase the transmission delay in our real-time control application.

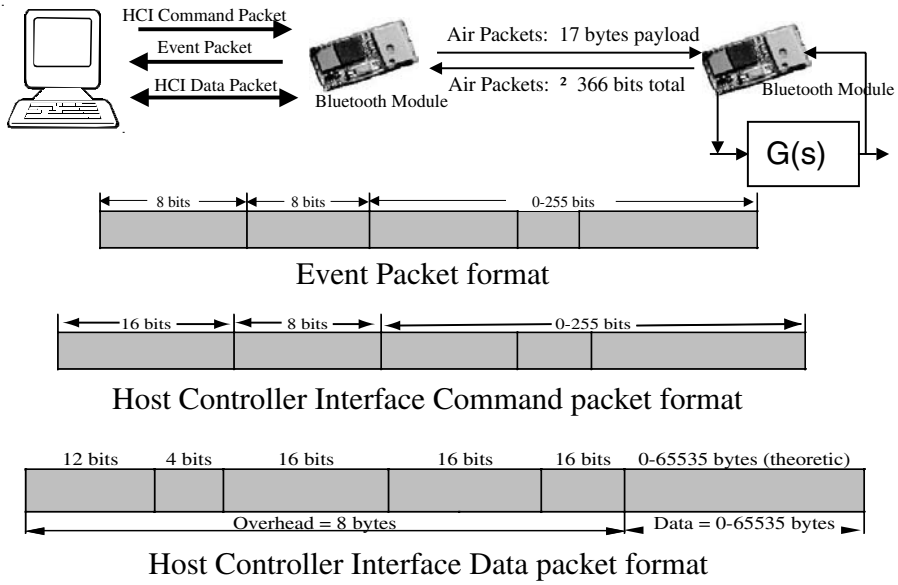


Fig. 10. Bluetooth packet structures. The top of the figure is a cartoon description of the pendulum-cart experimental setup. It shows the connection of the control computer (left) with the Bluetooth EBSK module, which has established a wireless link to a second EBSK module, which is in turn connected to our plant—the actual pendulum-cart apparatus.

HCI packets are passed to the EBSK firmware on the wireless module. If the packet type indicates a command packet, the firmware executes the command and returns an event packet. If the packet type indicates a data packet, the firmware parcels the data into a number of DM1 air packets which are transmitted over the wireless medium. The maximum size of the payload in a DM1 packet is 17 bytes. Each DM1 packet represents 355 bits transmitted over the wireless link, indicating an overhead of $366 - 136 = 230$ bits (28.75 bytes).

It has turned out in our application that, of the 17 bytes generated in the HCI layer, only 9 bytes are data which are useful to the application (state-information regarding the pendulum-cart system). At this rate, one sees that 9 bytes (72 bits) are carried in packets of 366 bits, meaning that $72/366 = 19.6\%$ of the DM1 data packet is useful to the application. If one were to send a payload of, say, 10 bytes of data useful to the application, the HCI layer would require a payload size of 18 bytes. This would cause fragmentation of the DM1 packets, requiring one with 17 bytes of payload and one with 1 byte of

payload. These packets would have sizes 366 and 186 bits, respectively. In this case, only $80/(366 + 186) = 14.4\%$ of the DM1 air packet carries information useful to the application. With the data being sent as two packets, there is an increase in the packets being dropped and thus being retransmitted. It also increases the total data transmitted and the consequent latency over the link as seen by the application. It is thus desirable to avoid fragmentation in the creation of DM1 packets, although, depending on the amount of real-time data that is essential to the application, it is not always possible to do so.

6 Experiments with a Real-Time Testbed

Recall that there is a relationship between a control system's sample rate and the transmission delay. For the pendulum-cart system, sampling intervals as large as 40ms (25Hz) can be tolerated. However, the transmission delay cannot be larger than 22ms, in which case sampling needs to be performed at intervals no longer than $\approx$ 22ms. When the transmission delays are significantly smaller, the sampling interval can be larger.

Timing diagrams similar to Fig. 8 apply to our experiment. The state of the plant is sent over the wireless link to the controller. The four states (five, when the RTT value is included) are each encoded by 16-bit unsigned short integers. Thus, a total of 8 or 10 bytes are sent across the wireless link. The previous section discussed the data budgets in each packet. Using this packet audit, it is possible to understand the data rates achievable over our Bluetooth link. First, an HCI packet is created in the control computer and transmitted down to the EBSK Bluetooth module using an RS232 cable. The RS232 data rate is 57.6kbps. Assuming a 17-byte HCI data packet (which maps to 18 bytes for an RS232 HCI packet due to the 1 byte packet type field), one gets a one-way transmission delay of $144/57.6 = 2.5$ ms between the computer and the Bluetooth module. Next these 17 bytes are parceled into a DM1 packet of size 366 bits. The air packet transmission delay, based on the data rate of 108kbps is $366/108 = 3.39$ ms. The RTT is computed based on four transfers of data over serial links and two transfers over the air (DM1 packets). This totals $4\times2.5 + 2\times3.39 = 16.78$ ms. The actual observed RTT is of the order of 20ms, with a standard deviation of 2 ms. Among the reasons for the difference in the predicted and actual values are the scheduling of data exchanges with the serial buffer by the operating system and the overhead due to processing the data at both ends of the system.

It is clear that under ideal operating conditions, stable operation of the inverted pendulum apparatus is possible. When there are dropped packets, however, the system may lose stability, and our observation in the lab was that the experiment could only be run for about a minute before the pendulum would fall. Many additional experiments using adaptive control techniques and model predictive control were carried out. Reports of these approaches are beyond the scope of this chapter. For more information about such approaches,

the reader is referred to [8]. For further reading on Bluetooth and 802.11 standards, see

`http://www.bluetooth.org`
`http://grouper.ieee.org/groups/802/11/index.html`
`http://www.ieee802.org.`

For additional details regarding supervisory control of multiagent systems connected through wireless networks, we refer to [9]. For general information regarding Bluetooth networks, the reader should see [2], and for a broad and detailed view of current research on networked control systems, the reader is referred to [1].

References

1. P.J. Antsaklis and J. Baillieul, Eds., Special Issue on Networked Control Systems, *IEEE Trans. on Autom. Control*, Vol. 49, No. 9, September 2004.
2. Jennifer Bray and C. Sturman, *Bluetooth: Connect Without Cables*, First Ed., Prentice Hall PTR, Upper Saddle River, NJ, ISBN: 0130898406; pp. 528, 2001.
3. W. Grega, K. Kolek, and A. Turnau, April 1997. *Digital Pendulum Control System*, Manual 33-005-3, Feedback Instruments Ltd., also *Digital Pendulum Control System*, Manual 33-930, Beta Release, Feedback Instruments Ltd., Crowborough, U.K.
4. R.N. Johnson, IEEE-1451.2 update, in *Sensors Online*, January, Volume 17, Number 1, http://www.sensorsmag.com/articles/0100/index.htm, 2000.
5. K. Li and J. Baillieul, Robust quantization and coding for multidimensional linear systems under data rate constraints, *43rd IEEE Conference on Decision and Control*, December 2004, WeB03.2.
6. N.J. Ploplys, P.A. Kawka, and A.G. Alleyne, Closed-loop control over wireless networks, *Control Systems Magazine*, June, Vol. 24, No. 3, pp. 58–71, 2004.
7. D. Qiao, S. Choi, and K.G. Shin, Goodput analysis and link adaptation for IEEE 802.11a wireless LANs, *IEEE Transactions on Mobile Computing*, Vol. 1, No. 4, October–December 2002, pp 278–292.
8. D.V. Raghunathan, *Real Time Control Over Data Networks with Constrained Communication Resources*, Boston University M.S. Thesis, Department of Aerospace and Mechanical Engineering, January 2002.
9. A.A. Suri, *Information Patterns in Formation Control of Autonomous Vehicles*, Boston University M.S. Thesis. Department of Aerospace/Mechanical Engineering, January 2004.
10. J. Tourrilhes, Dwell adaptive fragmentation: How to cope with short dwells required by multimedia wireless LANs, *Global Telecommunications Conference*, 2000, GLOBECOM '00, 11/27/2000–12/01/2000, San Francisco, CA, vol. 1, pp. 57–61.
11. H. Ye, G.C. Walsh and L.G. Bushnell, Real-time mixed traffic wireless networks, *IEEE Trans. on Industrial Electronics*, October 2001, Vol. 8, No. 5, pp. 883–890.

Bluetooth in Control

Bo Bernhardsson, Johan Eker, and Joakim Persson

Research, Ericsson Mobile Platforms, 221 83 Lund, Sweden
bo.bernhardsson@ericsson.com
johan.eker@ericsson.com
joakim.m.persson@ericsson.com

1 Introduction

Bluetooth is a short range radio technology, initially designed in 1994 at Ericsson Mobile Communications as a cable replacer for mobile devices. It was typically to be used for connecting mobile phones with PCs to synchronize and exchange data, or with accessories, such as a wireless hands-free device. Bluetooth was designed with a strong focus on low power and low cost. Ericsson together with Nokia, IBM, Intel, and Toshiba, formed a special interest group in 1998 to further develop and standardize the technology. In 1999 the open Bluetooth specification, version 1.0 was released. The name Bluetooth is from the Danish Viking king Harald Blåtand (Harald Bluetooth) who is famous for uniting Norway and Denmark and had a tooth that was discolored.

During 2003, there were approximately 69 million Bluetooth units shipped —a doubling of the corresponding number from 2002. Today, these chipsets are mostly deployed in cell phones, and the penetration of Bluetooth within mobile terminals (both for communication and for computing) is expected to grow significantly in the coming years. It can be expected that the technology will also become available in many other consumer products as the prices will continue to drop during the next couple of years.

While Bluetooth was designed for the office and consumer product markets, it has proven useful outside the initial target application area as well. Bluetooth has many qualities that are valuable in industrial automation applications: It is low power, it is tolerant against noisy environments, it has built in security, it has low latencies, etc.

Going wireless has some obvious advantages; without cables we have much greater freedom to physically distribute the nodes. It is possible to place actuators and sensors without worrying about the location of the control node. For example, we can place the sensor and actuator on a mobile object and still run the controller from a stand-still platform. However, going wireless will also result in less dependable systems. The likelihood of a wireless link containing bit errors is several orders of magnitude larger than for a wired link.

This means that the possibility of a message being corrupted and delayed due to retransmissions is much larger. The problem of unreliable links and time delays must be explicitly addressed in any wireless control application.

In a distributed control system the sensors, the actuators and the controller are physically displaced and communicate over a network. The main advantage of this setup is flexibility. It allows us to use the same control computer for several control loops and it relieves us from the potential problem of placing the controller close to the controlled process. However, a downside of using a distributed approach is that delays will be introduced in the control loops due to latencies in the network communication. The delays will lead to decreased phase margins in the control loops and potentially unstable systems.

This chapter discusses how Bluetooth may be used in automation and control applications. We will start by giving an overview of Bluetooth as such from an automation point of view, i.e., what kind of performance in terms of sampling rates, latency, etc., can one expect from an automation system utilizing Bluetooth? We will then use this knowledge and design some example wireless control systems.

2 Bluetooth

The Bluetooth wireless technology is specified in [3] and a good introduction is found in [8]. Its architecture follows a layered approach where each layer at one end talks to the corresponding layer at the other end. The principle is illustrated in Fig. 1. At the bottom is the *physical layer*, which defines the characteristics of the radio transmitter and receiver. On top of this, the *baseband layer* defines how packets are formed and controls the timing of the communication channel. Above the baseband layer we have the *link manager* (LM), which sets up links, negotiates features, and administers the running connections. This task is handled using the *LM Protocol* (LMP). Before data can be sent over the wireless link, large chunks of user data must be reformatted into packets suitable for handling at the baseband layer. At the receiving end, the reverse procedure is applied—small chunks of data are reassembled and released to higher layers in the correct order. This task is handled by the *Logical Link Control and Adaptation Protocol* (L2CAP).

It is natural to build the functionality defined by the layers described above into a module and incorporate this module into some gadget—a *host device*. Such modules are referred to as *Bluetooth controllers*. To facilitate communication between higher layer protocols running on the host and the lower layer protocols handled by the Bluetooth controller, a *Host Controller Interface* (HCI) has been defined. This interface describes how to access the functionality of the lower layer protocols executed by the Bluetooth controller.

Above the L2CAP, several higher layer protocols have been defined, for instance, the *Service Discovery Protocol* (SDP) and a serial connection protocol

denoted by *RFCOMM*. Furthermore, there exist several *profiles* that constitute "vertical slices" through the protocol stack. The profiles define what functionality is needed in order to perform a certain task. Examples of such profiles are the *Advanced Audio Distribution Profile* (A2DP) and the *Dial Up Networking* (DUN) profile .

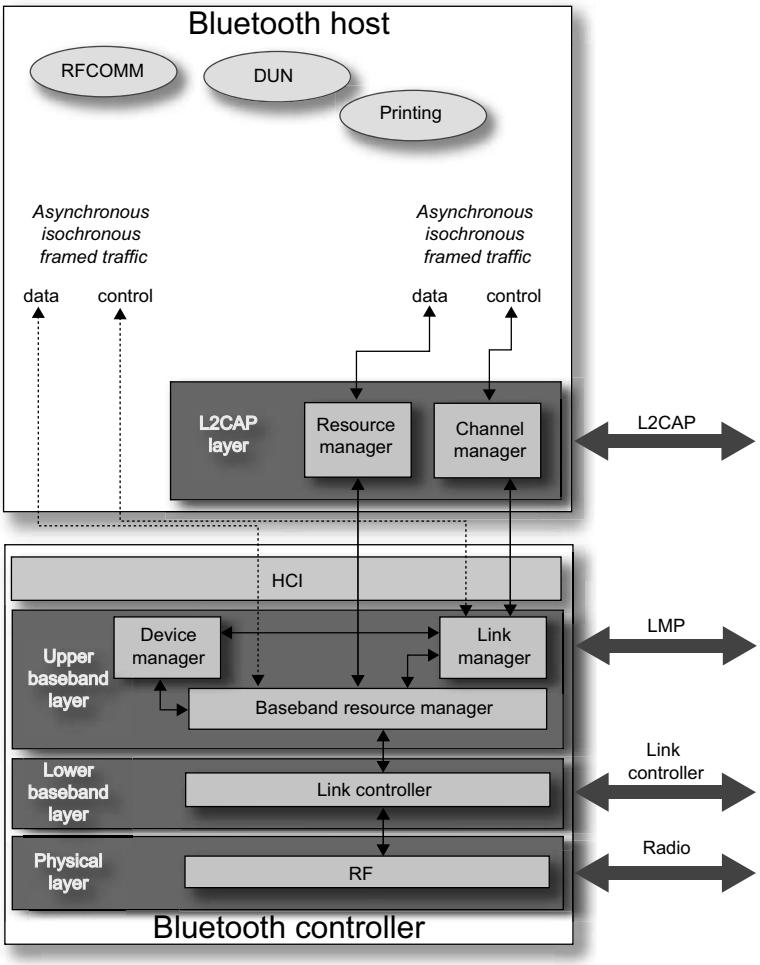


Fig. 1. A schematic view of the Bluetooth protocol stack

Bluetooth devices connect in a star topology (see Fig. 2) where the central node is denoted as *master* and the other nodes are denoted as *slaves*. All traffics flows between the master and the slaves (both directions are possible, of course)—traffic between slaves is not possible. The medium access is controlled by the master, who actively has to poll a slave before this slave

can send any data. Flexible timesharing is possible in the forward and reverse directions in order to accommodate asymmetric traffic flow between master and slave.

2.1 Physical layer

The radio works in the globally available 2.4 GHz *industrial, scientific, and medical* (ISM) band. The maximum output power depends on the power class of the device. Class 3 is restricted to 1 mW, class 2 is restricted to 2.5 mW, and class 1 is restricted to 100 mW. Using the nominal numbers on receiver sensitivity given in the Bluetooth specification, this will allow units in line of sight to communicate within a distance of at least 10 m when operating in class 3, and 100 m when operating in class 1.

In order to decrease susceptibility to interference, Bluetooth deploys *frequency hopping* (FH) spread spectrum technology. There are 79 channels used, each with a bandwidth of 1 MHz. During communication, the system makes 1600 hops per second evenly spread over these channels according to a pseudo-random pattern. The particular sequence of frequencies used is referred to as a *physical channel*. The idea is that if one transmits on a bad channel, the next hop, which occurs only 625 μs later, will hopefully be on a channel that is not interfered by any other radio source. In general, faster hopping between frequencies gives more spreading, which improves protection from other interferers. However, the improved performance comes at the cost of increased complexity. The hopping rate chosen for Bluetooth is considered to be a good trade-off between performance and complexity.

The signal is transmitted using binary *Gaussian frequency shift keying*. The raw bit rate is 1 Mb/s, but due to various kinds of protocol overhead, the user data rate cannot exceed 723 kb/s. Future versions of the Bluetooth specification are likely to increase the maximum achievable data rate by changing the modulation format and, possibly, also by increasing the symbol rate.

2.2 Baseband layer

This layer handles functionality related to link control, such as assembling packets into the format that is sent over the air, handling the *Automatic Repeat reQuest* (ARQ) protocol, performing channel encoding and decoding and encryption and decryption, and generating the FH sequence.

Connection setup

One of the greatest features of Bluetooth is the simplicity by which *ad hoc* networking works. This is the notion of a networking mode that does not require any particular infrastructure in the sense of base stations or access points being in place before communication can start. Instead, Bluetooth allows for

dynamic forming of networks on demand; any unit is capable of setting up (and tearing down) a local network when it is desirable. There is no need for certain *a priori* constellations among the Bluetooth devices, nor is there any difference between the possible networking roles the different units are capable of handling.

A unique 48 bit address is associated with each Bluetooth device. This address, denoted by *BD_ADDR*, is factory preset and cannot be changed by the user. It is used for identification purposes when connections are established. To address a particular device—also known as *paging* a device—the *BD_ADDR* of the addressee must be known to the caller. An *inquiry procedure* is defined that allows a Bluetooth device to find out the *BD_ADDR* of nearby units. This is accomplished by broadcasting a specific inquiry message and collecting device addresses from units that respond to this inquiry message. Clearly, the inquiry procedure is only needed for first-time connections as the inquirer can store the *BD_ADDR* of a responding device of interest for later use.

The *inquiry* message is repeatedly transmitted following a well-defined, rather short hop sequence of length 32. Any device that wants to be visible to others (also known as being *discoverable*) frequently scans the inquiry hop sequence for inquiry messages. This procedure is referred to as an *inquiry scan*. A scanning device will respond to inquiries with its *BD_ADDR* and the current value of its native clock. The inquiry message is anonymous and there is no acknowledgment to the response, so the scanning device has no idea who made the inquiry, or if the inquirer received the response correctly.

To reach a particular device, a *page* message is sent. This message is sent on another length 32 hop sequence determined from the 24 least significant bits of the *BD_ADDR* (which are denoted the *lower address part* (LAP)) of the target device. A device listens for page messages when it is in *page scan* state. The phase (i.e., the particular position) of the FH sequence is determined from the device's native clock. The paging device has knowledge of this from the inquiry response, thus it is possible for the paging device to hit the correct frequency of the paged device fairly quickly. As already stated, the inquiry part can be bypassed when two units have previously set up a connection and want to connect again. If a long time has passed since the previous connection, the clocks of the devices may have drifted, causing the estimate of the other unit's native clock to be inaccurate. The only effect of this is that the connection setup time may increase because of the resulting misalignment of their respective phase in the page hop sequence.

When a page response is received, a rough FH synchronization has been established between the pager and the paged device. By definition, the pager is the *master* and the paged device is the *slave*. Before the channel can be set up, some more information must be exchanged between the devices. The FH sequence, the timing, and the *channel access code* (CAC) are all derived from the master device. In order to fine-tune the FH synchronization, the slave needs the *BD_ADDR* and the native clock of the master. This information

is conveyed in a special packet sent from the master to the slave. With all information at hand at the slave side, the master and slave can switch from the page hopping sequence (defined by the slave) to the basic channel hopping sequence determined by the master's parameters.

Topology and the physical channel

Bluetooth supports a point-to-point connection or a point-to-multipoint connection. The former implies that the physical channel is shared among two devices, while the latter implies sharing the physical channel among more than two Bluetooth devices. The devices sharing one physical channel are forming a *piconet*. A piconet has exactly one master and at least one, but no more than seven slaves. Within a piconet, each slave will be given a three bit address denoted by *LT_ADDR*. The *LT_ADDR* is unique within the piconet and it is used by the master for addressing slaves rather than the much longer *BD_ADDR*.

Data can only be sent between the master and the slaves; there is no option for slave-to-slave traffic. Should such a traffic flow be desirable, either the master must actively relay it between the slaves (requiring support from some higher layer protocol), or, preferably, the slaves set up a piconet of their own. When multiple piconets cover the same area, it is possible for a device to participate in more than one of these simultaneously using timesharing. A device may act as slave in several piconets, but it is only possible to be master in a single piconet. A group of piconets where there exist common nodes in the networks is called a *scatternet*. A few examples of piconet constellations are depicted in Fig. 2.

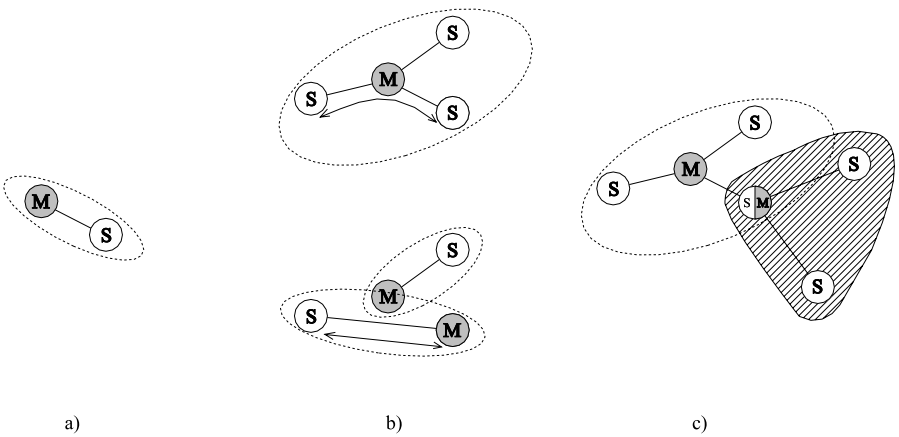


Fig. 2. Some examples of different piconet constellations: (a) point-to-point, (b) master relaying versus separate piconets, and (c) timesharing to achieve a scatternet

On the physical channel, a *time division duplex* (TDD) scheme is used to share the medium. The time line is divided into slots of 625 μs duration. Master-to-slave traffic can only start at even slot numbers, while slave-to-master traffic can only start at odd slot numbers. As the packets that are transmitted are allowed to occupy one, three, or five slots, and the number of used slots need not be the same in both directions, one can easily accommodate asymmetric traffic streams. It is perfectly fine to use "empty" slots (NULL packets) if no user data is available. The NULL packets do carry some information related to the ARQ protocol, thus they need to be transmitted even though actual user data is missing. In Fig. 3, the principle of the TDD scheme is illustrated.

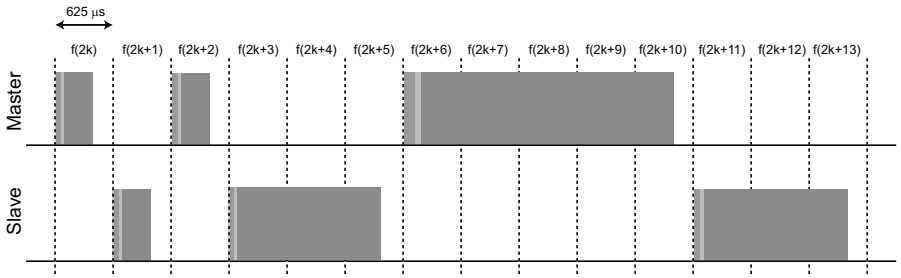


Fig. 3. The TDD scheme used in Bluetooth

The master grants channel access to individual slaves by addressing them. A slave is not allowed to transmit unless it is explicitly polled by the master. By polling unequally among the slaves, the total capacity of the piconet can be distributed quite freely among the links that share it. It should be noted that one major idea behind Bluetooth is that the overall throughput within a small area is higher when there are several small piconets operating in parallel, rather than a few large ones. Even though the number of collisions increases with more piconets, the aggregated throughput supersedes that which can be reached when many units share a collision-free common channel.

Each unit has a 28 bit native clock running at 3200 Hz. The FH sequence used is different for each piconet and it is derived from the master's *BD_ADDR* and its native clock. The slots are numbered according to the 27 most significant bits of the master clock. For each new slot, a new frequency is calculated. However, for multi-slot packets the frequency stays constant during the entire transmission. The FH sequence for each piconet is cyclic with a period time of approximately 23 hours.

Packet formats

A Bluetooth packet consists of an *access code*, a *header*, and a *payload*. The 72 bit access code is uniquely derived from the master's *BD_ADDR*. As this

implies different access codes for every piconet, it provides a method for slaves to filter out packets that do not belong to their piconet. The header consists of 18 bits of information (with an $R = 1/3$ repetition code applied to increase robustness, thus 54 bits are transmitted) used for administrative purposes such as addressing within the piconet, ARQ protocol parameters, flow control, etc. Finally, the payload may carry 0–2744 bits of user data. A 16 bit cyclic redundancy check (CRC) is used for error detection on the payload for some packet types. It is also possible to apply one of two different error correcting codes to this part—either the $R = 1/3$ repetition code used for the header or an $R = 2/3$ shortened Hamming code. Fig. 4 illustrates the packet format.

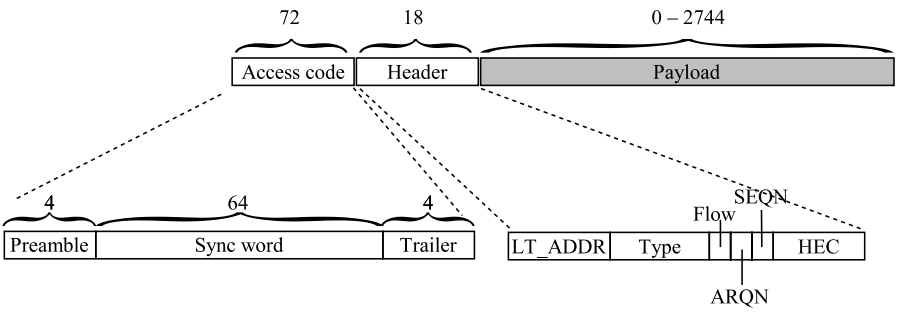


Fig. 4. A baseband packet

As clock drift between different units is inevitable, each slave must track and keep an updated offset of its own clock to the master's clock. This offset is calculated based on the arrival time of packets sent from the master. In practice, the receiver computes the offset by looking at where the peak correlation occurs between the received (most likely noisy) access code and a sliding window of the well-defined access code of the piconet.

Bluetooth can support a mixture of traffic types on the same piconet, and even on the same link. Best effort traffic has no requirement on latency or delay, but high demands on correctness of delivered data. This is often referred to as *asynchronous* traffic. In Bluetooth, this traffic is handled using *asynchronous connection-oriented link* (ACL[1]) for which ARQ is mandated and error correcting coding is optional. The ACLs are denoted by DH1, DH3, and DH5 for 1, 3, and 5 slot packets, respectively. If the optional error correction code is used (indicated by changing the letter H of the type name to an M), the amount of user data decreases, which is manifested as a decreasing data rate. Table 1 summarizes what data rates can be accomplished for different ACL packets.

[1]This acronym used to be interpreted as *asynchronous connectionless*, but in version 1.2 of the Bluetooth specification this has been changed, as an ACL link really is a connection-oriented logical transport. To be consistent with older documentation, the acronym itself is unchanged.

Type	Payload (information bytes)	FEC	CRC	Symmetric max. rate (kb/s)	Asymmetric max. rate (kb/s) Forward	Reverse
DM1	0–17	2/3	Yes	108.8	108.8	108.8
DH1	0–27	No	Yes	172.8	172.8	172.8
DM3	0–121	2/3	Yes	258.1	387.2	54.4
DH3	0–183	No	Yes	390.4	585.6	86.4
DM5	0–224	2/3	Yes	286.7	477.8	36.3
DH5	0–339	No	Yes	433.9	723.2	57.6
AUX1	0–29	No	No	185.6	185.6	185.6

Table 1. Summary of ACL packets and their achievable data rates

For real-time, two-way communication, the round-trip delay must be minimized and variations in the interarrival time of data samples should also be kept small. This is referred to as *synchronous* traffic. Synchronous traffic can be handled using the *synchronous connection-oriented* (SCO) link, for which slots are reserved on a regular basis. The SCO link is run without ARQ, but error correcting codes can be applied.

In the version 1.2 release of the Bluetooth specification, one more synchronous link has been defined—the eSCO link. As is the case for SCO links, the eSCO link provides constant rate data services by carrying fixed-sized packets on reserved slots over the physical channel. The difference lies in the flexibility provided by eSCO: data rates can be chosen more freely, and it is more reliable as a limited number of retransmissions can take place in between the reserved slots. The payload length is set during LMP eSCO setup and remains fixed until the link is removed or re-negotiated. The retransmission window immediately follows the reserved slots. The principle of single-slot eSCO packets is depicted in Fig. 5.

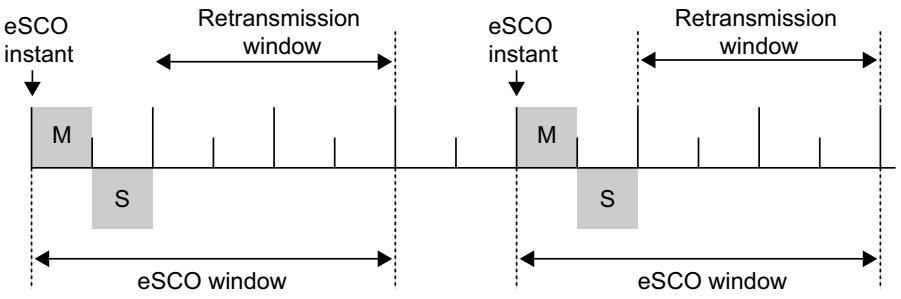


Fig. 5. Details for single-slot eSCO windows

Table 2 lists the available data rates for synchronous packets. As indicated there, the synchronous HV1, HV2, and HV3 packets are fixed sized without ARQ, while the eSCO links (EV3, EV4, and EV5) are flexible in size as well as

in utilizing retransmission. It can be noted that eSCO traffic uses a secondary *LT_ADDR*, which allows mixing of ACL and eSCO traffic to the same unit with independent ARQ handling for both traffic streams.

Type	**Payload (information bytes)**	**FEC**	**CRC**	**Symmetric max. rate (kb/s)**
HV1	10	1/3	No	64
HV2	20	2/3	No	64
HV3	30	No	No	64
DV	10 + (0–9)*	2/3*	Yes*	64 + 57.6*
EV3	1–30	No	Yes	96
EV4	1–120	2/3	Yes	192
EV5	1–180	No	Yes	288

Table 2. Summary of synchronous packets and their achievable data rates. The *-marked items of the DV packet are only relevant to the data part of the payload.

2.3 The LMP

It is the LM that is responsible for the control of the Bluetooth link. That includes all tasks related to the setup, detach, or configuration of a link. The LM is also responsible for exchanging security-related messages. The LMs in different units exchange control messages using the LMP. A large set of control messages or LMP *protocol data unit*s (PDUs) have been defined. Many of these are security related and some PDUs are used to carry the information needed at pairing, authentication, and for enabling of encryption. LMP PDUs are always carried in the payload of one of two different types of single-slot packets. In order to distinguish LMP packets from conventional packets, a special type of code is used in the packet header of all LMP messages.

2.4 The L2CAP

The L2CAP takes care of datagram segmentation and re-assembly, multiplexing of service streams, and quality of service issues. The L2CAP constitutes a filter between the Bluetooth independent higher layers running on the host and the lower layers belonging to the Bluetooth module. For instance, TCP/IP traffic packets are too large to fit within a baseband packet. Therefore, such packets will be cut into smaller chunks of data before they are sent to the baseband for further processing. On the receiving side, the process is reversed; baseband packets are re-assembled into larger entities before they are released to higher layers.

2.5 The HCI

The HCI defines how to communicate between the upper layers and lower layers in the Bluetooth communication stack. The HCI command packets can be divided into six different subgroups:

- link control commands
- link policy commands
- host controller and baseband commands
- read information commands
- read status commands
- test commands

The link control commands are used to control the link layer connections to other Bluetooth devices. Control of authentication and encryption as well as keys and pass-key commands belong to this subgroup. The policy commands are used to control how the link manager manages the piconet. The host controller and baseband commands are used to read and write into several different host controller registers. This includes reading and writing keys and pass-keys to/from the host controller as well as reading and writing the general link manager authentication and encryption policy.

By defining an HCI, it is possible to logically separate the radio and baseband related functions from higher layer protocols. This is beneficial since the latter can be run on a host processor separated from the radio hardware. The radio can then be implemented as a separate module accessible through a well-defined interface. For instance, a laptop may implement higher protocol layers in software and incorporate the Bluetooth radio module, either directly on the motherboard, or as a dongle connected to one of the standard interfaces such as the universal serial bus (USB) or the PC card bus. However, not all Bluetooth implementations run the lower and higher layer processing on different processors. Integrated implementations are also possible. Consequently, the HCI is an optional feature, and only products that benefit from the separation use it.

2.6 Security

Unlike many other wireless communication systems, Bluetooth has been designed with security in mind. Consequently, the specification has incorporated means for security right from the beginning. The intention is to provide what commonly is called *link level security*. The user should not feel that privacy is more of a concern with the Bluetooth link than with a wired connection. This is accomplished by providing *authentication* and *encryption*. Authentication is the process of verifying the existence of a shared secret denoted by the *link key* in two devices, whereas encryption is the process of hiding the information that is exchanged for all but the holders of the *encryption key*.

The latter usually implies two devices (master and slave), but there is an option for broadcast encryption that allows for up to seven slaves listening to one master broadcasting encrypted messages.

The security provided by the Bluetooth core is built upon the use of symmetric-key cryptographic mechanisms for authentication, link encryption, and key generation. A number of different key types are used in connection with these mechanisms. Link keys are primarily used for authentication, but they are also used to derive the encryption key that controls the encryption of the data sent via a link. The size of all keys is limited to at most 128 bits due to export regulations. The link key is created during the *pairing* of two devices, and its existence is required before any security functionality can be applied. Usually the pairing procedure will involve some manual user interaction in order to enter a pass-key that is used in the process of generating the link key. Each device must keep records of the link keys and their associated *BD_ADDR* that were involved in the respective pairing. Preferably this list is stored in non-volatile memory. Then, the manual interaction is only necessary for first-time connections as the stored link key will be used for subsequent connections. A thorough description of Bluetooth security can be found in [7].

3 Distributed Control Systems

Control systems that use imperfect or shared resources such as CPU, memory, or a communication network are often hard to analyze and must be designed with care. One should be aware of how to recognize problems that are due to communication errors and time delays introduced by the data scheduling. Fortunately, there are special design methods that can take the communication network behavior into account and thus minimize performance degradation due to communication errors and delays. Fig. 6 shows a distributed control system.

An air interface such as Bluetooth introduces both random transmission errors and delays. For slow, non-time-critical processes, this problem can be neglected. If there is time for many retransmissions the error rate can be made sufficiently low. For faster processes, the communication network can degrade performance. By choosing the control algorithm, scheduling mechanism, and retransmission scheme one can limit the performance degradation.

If the delay variations can be measured, there are control algorithms that compensate for the delay and minimize the performance degradation. For a Bluetooth control system where the controller node is chosen as master, all time delays can be known for the controller when acknowledge mode transmission is used. In other cases, time-stamps can be attached to the signalled information to make delay estimation possible; however, this requires synchronized clocks in the nodes.

The standard rule of thumb for choice of sampling rate [2], $\omega h = 0.2 - 0.6$ where ω is the cross-over frequency of the closed loop system, gives an

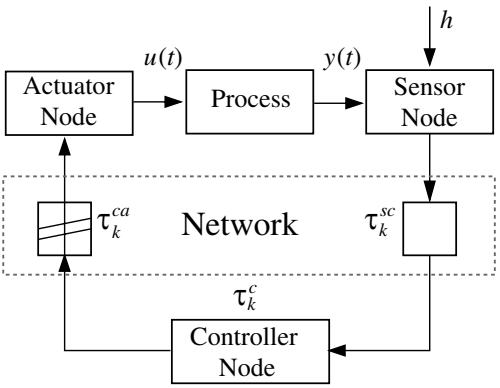


Fig. 6. Distributed control system with three delays: sensor-to-controller delay τ^{sc}, control computation delay τ^{c}, and control-to-actuator delay τ^{ca}

indication of when time delay variations can be neglected in the design process. For example, a constant delay of one sample decreases the phase margin of the system by 11 to 34 degrees, if the sampling interval has been chosen according to the rule of thumb above. Assuming that we can neglect time delay variations when they are smaller than 10–20 percent of the sampling time, we get the condition $f\tau < 0.01$ (where f is the cross-over frequency in hertz) for when we most likely can neglect the time delay variations.

3.1 Choice of control algorithm

Depending on which control algorithm is used, different patches for communication errors and delays can be applied. In this section we will briefly address these issues for both PID controllers and state feedback controllers. A standard proportional-integral-derivative (PID) controller is given as

$$u(t) = Ke(t) + \frac{K}{T_i}\int e(s)ds + KT_D\frac{de(t)}{dt},$$

where K, T_i, and T_D are parameters and $e(t)$ is the control error. A good discussion on the implementation of the PID controller is found in [1]. A common discrete time form of the PID is

$$\begin{aligned} u(k) &:= P(k) + I(k) + D(k) \\ P(k) &:= K(r(k) - y(k)) \\ I(k+1) &:= I(k) + (Kh/Ti)e(k) \\ D(k) &:= \frac{T_D}{T_D + Nh}D(t_{k-1}) - \frac{KT_DN}{T_D + Nh}(y(t_k) - y(t_{k-1})), \end{aligned}$$

where h is the sampling interval. For delay variations in the measurements for the PID controller, a straightforward strategy is to replace the sample interval

with $h = t_{k+1} - t_k$, where t_k is the time when measurement k was taken, and to change the implementation of integral and derivative part accordingly. Delay variations from the controller to the actuator are harder to compensate for. One possibility is to put some intelligence in the controller node, as described in Section 3.1.

A simple patch for lost sensor samples is to use the D-part to extrapolate the error, i.e., $e_p(k+1) := e(k) + h\dot{e}(k)$. The control signal then becomes $u(k) = P(k) + I(k) + D(k)$, where

$$\begin{aligned} P(k+1) &:= Ke_p(k+1) \\ I(k+1) &:= I(k) + (Kh/Ti)e_p(k+1) \\ D(k+1) &:= D(k). \end{aligned}$$

For lost actuator samples the actuator can apply the control signal from the previous sample. More elaborate schemes are also possible, but they require more intelligence in the actuator node.

We will now take a look at state feedback control. Let the system to be controlled be given in the following discrete form:

$$\begin{aligned} x(k+1) &= \Phi x(k) + \Gamma u(k) \\ y(k) &= Cx(k) + Du(k). \end{aligned}$$

For state feedback control, the case of constant delay is easy to cope with in the design. The sampled-data description of the plant including the control delay will simply be of higher order, and the design can be made using standard techniques. The control law has, for time delays smaller than one sample period, the form

$$u(k) = -L \begin{bmatrix} x(k) \\ u(k-1) \end{bmatrix},$$

where L is a constant feedback gain vector, $x(k)$ is the state vector, and $u(k-1)$ is the old control signal.

The case of randomly varying delays is more difficult. A simple but non-optimal solution is to perform static delay compensation for the mean delay.

The optimal LQ-controller for random sensor-to-controller and controller-to-actuator delays was derived in [9]. The controller performs dynamic delay compensation by adjusting the feedback gain according to the actual delays. The optimal control law has the form

$$u(k) = -L(\tau_k^{sc}) \begin{bmatrix} x(k) \\ u(k-1) \end{bmatrix}, \tag{1}$$

where the feedback gain vector L is now a function of the sensor-to-controller delay τ_k^{sc} in the current sample.

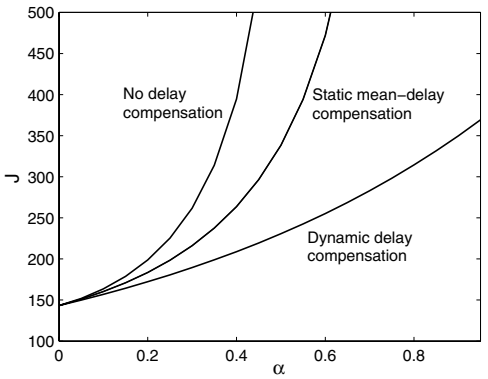


Fig. 7. Values of the cost function J for a pendulum controller under three different schemes. In this plot, the control delay is uniformly distributed on $[0, \alpha h]$.

Fig. 7 shows the cost J[2] as a function of the amount of stochastic delay for three different control schemes. When the delay becomes very long, the dynamically compensating controller is still stable ($J < \infty$), while the other control schemes become unstable.

Controllers designed via state feedback with state estimation from a Kalman filter are also easily patched for sensor communication errors. The Kalman filter can be run in open loop using the model of the process. For optimal linear quadratic Gaussian (LQG) design, the time-varying Kalman filter gains can be calculated using the standard equations, setting $C(t) = 0$ in the equations whenever the sensor measurement is unavailable.

Optimal compensation of actuator communication failures is harder, since setting $B(t) = 0$ in the state feedback equations requires these failures to be known in advance. If the actuator node applies a known control signal whenever communication is lost, a patch is to update the Kalman filter accordingly and use an adjusted feedback matrix L. The optimal feedback matrix for the control system where the zero control signal is applied for lost samples is

$$L = (Q_{22} + B^T S B)^{-1}(B^T S A + Q_{12}),$$

where the matrix S is given as the positive solution (if it exists) of the following non-standard Riccati equation:

$$S = Q_{11} + A^T S A - (1-p) \cdot (A^T S B + Q_{12})(Q_{22} + B^T S B)^{-1}(A^T S B + Q_{12}),$$

where p denotes the communication error rate (the Q-matrices define the loss function, as in [2]). For low error rates the standard Riccati equation can be used with only small error degradation.

[2]The cost function J is given as: $J = \lim_{T\to\infty} \frac{1}{T} \mathrm{e}\left\{ \int_0^T \left(x^T Q_1 x + u^T Q_2 u \right) dt \right\}$, for some Q_1 and Q_2.

A common alternative implementation is that the actuator node applies the previous control signal whenever samples are lost. This case can however be transformed to the previous case by rewriting the controller/system interface in incremental form, i.e., as $u_k = u_{k-1} + v_k$, where v_k is treated as the new control signal.

Example 1: Choosing a transmission scheme

To illustrate that choice of retransmission scheme can influence control performance we will analyze a simple system that captures the main features.

Let us assume that the sensor node sends measurements over an error-prone communication network to a controller node. The controller node is assumed to be co-located with the actuator node, i.e., we assume that no transmission between controller node and actuator node is necessary. We also assume that the CPU time to compute the control signal is available. At time k the sensor node samples the signal $y(k)$ and transmits it over the communication network in acknowledged mode. If the transmission is successful, the signal $y(k)$ is available for the controller node when control signal $u(k+1)$ is calculated. If the transmission fails, the sensor node tries to retransmit. We assume there is time for one retransmission before time $k+1$.

Assume that the open loop system is given by

$$\begin{aligned} x(k+1) &= ax(k) + u(k) + d(k) \\ y(k) &= x(k), \end{aligned}$$

where $d(k)$ is a white noise signal with unit variance. The loss function that describes control performance is chosen as $J = E(x^2)$ where E denotes mean value. This linear quadratic control problem has as a solution the optimal control signal (deadbeat control)

$$u(k) = -a\hat{x},$$

where $\hat{x}$ denotes the best estimate of the state $x(k)$ given the information available by the controller node at time k.

We will compare two retransmission schemes.

1. eSCO with one retransmission: If transmission and retransmission both fail, then no more retransmissions will be tried. Instead the sensor will be resampled at time $k+1$ and the sensor node will start transmission of $y(k+1)$.
2. Infinite number of retransmissions allowed: each message is retransmitted until transmission succeeds.

The optimal state estimate can be calculated using standard techniques for both scheme 1 and scheme 2.

Scheme 1: This is a standard time-varying LQG problem. The optimal control signal is given by $u(n) = -a\hat{x}$, where $\hat{x}$ denotes the optimal state estimate with the information available at time n, given recursively by

$$\begin{aligned}\hat{x}(n) &= ay(n-1) + u(n-1), \qquad \text{if } y(n-1) \text{ is available at time } n\\ \hat{x}(n) &= a\hat{x}(n-1) + u(n-1) = 0, \quad \text{if } y(n-1) \text{ is not available.}\end{aligned}$$

It turns out that it is impossible to stabilize the system if $|a|q > 1$, where q is the probability that a transmission is erroneous. This gives a bound of the quality needed of the communication system. If $|a|q < 1$ then the optimal controller achieves the control performance $J = 1 + a^2P$ where $P := E(x - \hat{x})^2 = 1/(1 - (aq)^2)$.

Scheme 2: The optimal state estimate is slightly more difficult to calculate. If $y(n-m)$ is the latest measurement available at time n the state estimate $\hat{x}(n)$ is given recursively by

$$\begin{aligned}\hat{x}(n) &= a\hat{x}(n-1) + u(n-1)\\ \hat{x}(n-1) &= a\hat{x}(n-2) + u(n-2)\\ &\vdots\\ \hat{x}(n-m+1) &= ay(n-m) + u(n-m).\end{aligned}$$

It can be shown that with this retransmission scheme the system can be stabilized only when $|a|q/(1-q) < 1$. The control performance is then given by $J = 1 + a^2P$ where $P = 1/(1 - (aq/(1-q))^2)$. It can be concluded that retransmission scheme 1 is preferable, since it allows larger error rates and gives better control performance. The two loss functions are compared for the case $a = 1.5$ in Fig. 8.

Example 2: The Furuta pendulum demonstrator

Dynamic delay compensation works better the shorter the controller-to-actuator delay is compared to the sensor-to-controller delay. If the controller is located *at* the actuator, the delay in the current sample can be known exactly and the correct gain can be applied. Modifying the setup is of course not a real solution, but using an I/O with only very limited computational power, this ideal behavior can be emulated closely. The idea is illustrated on the pendulum controller. This example is taken from [6], which gives a more in-depth presentation of the experiment.

The distributed control configuration is shown in Fig. 6. The feedback loop consists of three parts: the sensor, the controller, and the actuator. The sensor node is time-driven (with the frequency $f = 1/h$) while the controller and actuator nodes are event-driven. The sensor samples the process periodically and sends the measurement values to the controller. Upon receipt, the controller calculates a new control signal and sends it to the actuator node

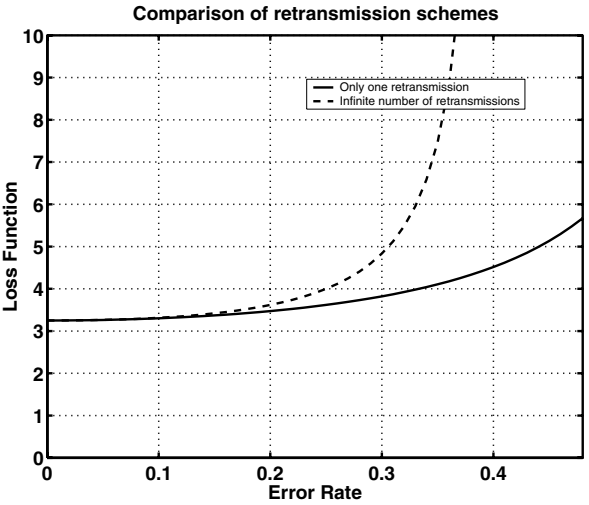


Fig. 8. Values of the cost function J for deadbeat control of an unstable system under different retransmission schemes

which outputs the value. In our setup, the sensor node and the actuator node are located in the same hardware unit, called the remote I/O, connected to a single Bluetooth unit.

In this experimental setup, the controller node is the Bluetooth master and the remote I/O is a slave. It is implemented with two PCs running Linux and the Harald Java Bluetooth stack. The theoretical minimum round-trip delay in Bluetooth is $2 \cdot 625$ μs. However, the length of the data packets, the Bluetooth protocol stack, and the communication between the Bluetooth hardware and the computer hardware bring the minimum round-trip delay up to 17 ms in our implementation. Still, this is fast enough to control the inverted pendulum, which is a process with fast and unstable dynamics. All messages are time-stamped, allowing for straightforward calculation of transmission delays in the receiver node.

The Furuta pendulum used is shown in Fig. 9. For a description of the Furuta pendulum see [10]. The objective is to control the pendulum angle θ and the arm angle ϕ to zero by applying the torque u to the rotating arm.

The full state vector is directly measurable on the process. The sampling interval for digital control was chosen as $h = 60$ ms. Discrete state-feedback controllers were designed based on a linear-quadratic (LQ) formulation where the control should minimize the cost function J.

The optimal feedback gain depends on the distribution of the round-trip delay and is computed using the formulas in [9]. The resulting control parameters are shown in Fig. 10. From the figure it is clear that the optimal gain can be closely approximated by a linear function of the delay τ. Hence, (1) may be approximated as

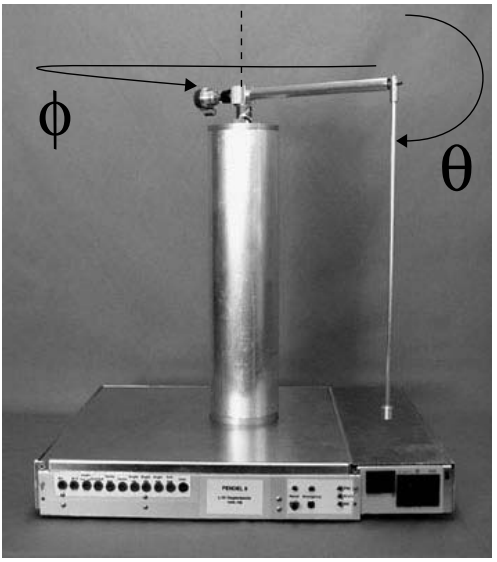


Fig. 9. The rotating inverted pendulum used in the experiments. The objective is to stabilize the pendulum in the upright position $\theta = 0$

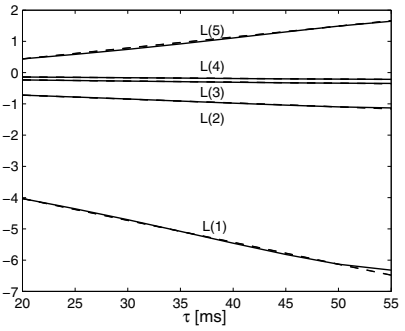


Fig. 10. Optimal gain vector (full lines) and linear approximation (dashed lines)

$$u(k) \approx -L_0 \begin{bmatrix} x(k) \\ u(k-1) \end{bmatrix} - \frac{dL}{d\tau} \begin{bmatrix} x(k) \\ u(k-1) \end{bmatrix} \tau.$$

In the controller node, $x(k)$ and $u(k-1)$ are known, but τ is still unknown. However, the controller can precompute

$$\hat{u}(k) = -L_0 \begin{bmatrix} x(k) \\ u(k-1) \end{bmatrix} \quad \text{and} \quad \lambda(k) = -\frac{dL}{d\tau} \begin{bmatrix} x(k) \\ u(k-1) \end{bmatrix}.$$

These two scalars are then sent to the remote I/O. If the I/O keeps track of the round-trip delay τ, it can perform the simple adjustment

$$u(k) = \hat{u}(k) + \lambda(k)\tau$$

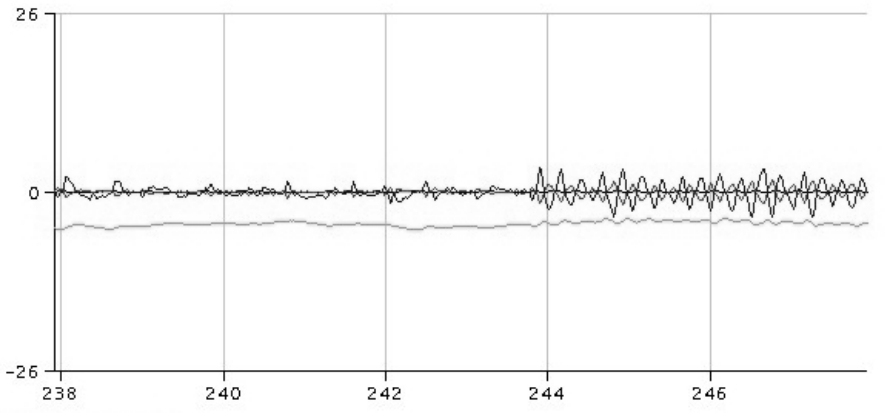


Fig. 11. Experiment on the Bluetooth-pendulum setup with random communication delays. At time 244, the dynamic delay compensation is turned off, and the pendulum starts to oscillate. A large part of the increase in the cost is due to unmodeled friction

before applying the control signal to the process.

The suggested delay compensation scheme was simulated against the nonlinear pendulum model. The pendulum was disturbed by white process noise and the round-trip delay in the control loop varied between 20 and 55 ms according to a given distribution. This was compared to a controller which was designed for a constant delay of 20 ms (the nominal round-trip delay). For this particular control problem, the dynamic delay compensation scheme was able to reduce the cost J by about 30 %.

The dynamic delay compensation was also tried on the distributed Bluetooth pendulum demonstrator. The random delays were not only due to actual disturbances, but were also injected in the control loop on purpose. The result of an experiment is shown in Fig. 11. Due to unmodeled friction, the pendulum easily started to oscillate when the delay compensation was turned off.

3.2 Tools and methods

Exact stability analysis of a system with time delay is well known if the delay Δ is constant or varying according to a known pattern. Here the delay Δ includes the sensor delay, sensor-to-controller transmission time, control computation delay, controller-to-actuator transmission delay, actuator delay, and the delay inside the physical system that is controlled.

The phase margin ϕ_m indicates how much constant time delay Δ can be added in the loop before the closed loop system becomes unstable; according to the Nyquist theorem it is given by $\Delta_{max} < \phi_m/\omega_c$, where ω_c is the cut-off frequency where $|CP| = 1$. The analysis of unknown, time-varying delay is significantly more difficult. If the time delay is known to belong to the interval

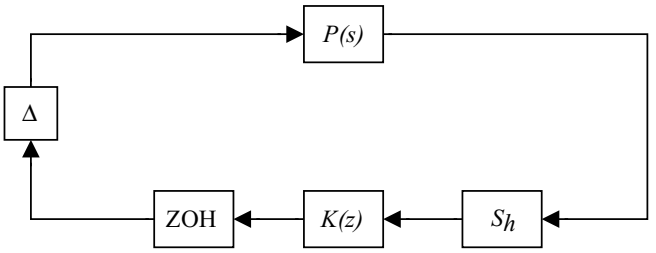


Fig. 12. A computer controlled system with continuous-time plant $P(s)$, periodic sampler S_h, discrete-time controller $K(z)$, zero-order hold sampling circuit, and a time varying delay Δ

$[\Delta_{min}, \Delta_{max}]$, a standard technique is to let a certain delay Δ_1 be included in the model of the system, and to design the system with this delay taken into account. Here typical choices are $\Delta_1 = \Delta_{min}$, Δ_{max}, or Δ_{avg} (the average time delay). None of these choices can be guaranteed to optimize performance; cases can be found where the closed loop system is stable for both the minimal and maximal time delay but becomes unstable when the delay is varying.

The *jitter margin* was introduced in [5] to describe how much time-varying delay can be tolerated before the control loop can become unstable. It is only a sufficient criterion, but gives a guarantee that the system is stable for any time delay variation in the given interval, including random delays.

Theorem 1. ***(Jitter Margin)*** *The closed loop system in Fig. 12 is stable for any time-varying delays $\Delta \in [0, Nh]$, where N is a real number, if*

$$\left| \frac{P_{alias}(\omega)C(e^{i\omega})}{1 + P_{ZOH}(e^{i\omega})C(e^{i\omega})} \right| < \frac{1}{\tilde{N}|e^{i\omega} - 1|}, \quad \forall \omega \in [0, \pi],$$

where $\tilde{N} = (\lfloor N \rfloor^2 + 2\lfloor N \rfloor g + g)^{1/2}$, and $g = N - \lfloor N \rfloor$; $P_{ZOH}(z)$ is the zero-order hold discretization of $P(s)$, and

$$P_{alias}(\omega) = \left(\sum_{k=-\infty}^{\infty} |P(i(\omega + 2\pi k)/h)|^2 \right)^{1/2}.$$

Good control design is greatly simplified by a set of proper tools. The two MATLAB-based tools TrueTime and JitterBug are presented in [4]. JitterBug allows analytic computation of quadratic performance criteria for linear control systems under various timing conditions. The tool can also compute the spectral density of the signals in the system. Using the toolbox, one can easily and quickly assert how sensitive a control system is to delay, jitter, lost samples, etc., without resorting to simulations. TrueTime is a simulation tool built on top of MATLAB, which facilitates simulation of the temporal

behavior of networked, multi-tasking control systems. It supports simulation of different communication networks and protocols, and their influence on the performance of networked control loops.

4 Summary

The application of wireless technology, such as Bluetooth, in the world of automation introduces a new level of freedom, but also a new set of problems. It is clear that wireless is much less reliable than wired communication. Both the variations in latency and the probability of losing an occasional sample are much greater. Therefore, special care must be taken to design control systems that are robust towards timing variations and also support graceful degradation in situations when connections are lost.

In this chapter on Bluetooth in control we have given an overview of the Bluetooth technology and an introduction to wireless control. A number of methods and tools have been briefly discussed.

References

1. K. J. Åström and T. Hägglund. *PID Controllers: Theory, Design, and Tuning*. Instrument Society of America, Research Triangle Park, North Carolina, 1995.
2. K. J. Åström and B. Wittenmark. *Computer-Controlled Systems*, 3rd edition. Prentice Hall, Upper Saddle River, NJ, 1997.
3. Bluetooth Special Interest Group (SIG). *Specification of the Bluetooth System, Version 1.2, Core System Package*, November 2003.
4. A. Cervin, D. Henriksson, B. Lincoln, J. Eker, and K.-E. Årzén. How does control timing affect performance? *IEEE Control Systems Magazine*, 23(3):16–30, June 2003.
5. A. Cervin, B. Lincoln, J. Eker, K.-E. Årzén, and G. Buttazzo. The jitter margin and its application in the design of real-time control systems. In *Proceedings of the 10th International Conference on Real-Time and Embedded Computing Systems and Applications - RTCSA*, Gothenburg, Sweden, August 2004.
6. J. Eker, A. Cervin, and A. Hörjel. Distributed wireless control using Bluetooth. In *Proceedings of the IFAC Conference on New Technologies for Computer Control*, Hong Kong, P.R. China, November 2001.
7. C. Gehrmann, J. Persson, and B. Smeets. *Bluetooth Security*. Artech House, Norwood, MA, 2004.
8. J. Haartsen and S. Mattisson. Bluetooth—New low-power radio interface providing short-range connectivity. *Proceedings of the IEEE*, 88(10):1662–1676, 2000.
9. J. Nilsson, B. Bernhardsson, and B. Wittenmark. Stochastic analysis and control of real-time systems with random time delays. *Automatica*, 34(1):57–64, 1998.
10. M. Yamakita, K. Furuta, K. Konohara, J. Hamada, and H. Kusano. VSS adaptive control based on nonlinear model for TITech pendulum. *IEEE*, 3:1488–1493, 1992.

Embedded Sensor Networks

John Heidemann[1] and Ramesh Govindan[2]

[1] University of Southern California/Information Sciences Institute, johnh@isi.edu
[2] University of Southern California, Computer Science Department, ramesh@usc.edu

1 Introduction

An *embedded sensor network* is a network of embedded computers placed in the physical world that interacts with the environment. These embedded computers, or *sensor nodes*, are often physically small, relatively inexpensive computers, each with some set of sensors or actuators. These sensor nodes are deployed *in situ*, physically placed in the environment near the objects they are sensing. Sensor nodes are networked, allowing them to communicate and cooperate with each other to monitor the environment and (possibly) effect changes to it. Current sensor networks are usually stationary, although sensors may be attached to moving objects or may even be capable of independent movement. These characteristics: being embedded, and being capable of sensing, actuation, and the ability to communicate, define the field of sensor networking and differentiate it from remote sensing, mobile computing with laptop computers, and traditional centralized sensing systems.

Although research in sensor networks dates back to the 1990s or earlier, the field exploded around the year 2000 with the availability of relatively inexpensive (sub-$1000) nodes, sensors, and radios. As of 2004, sensor networking is a very active research area with well-established hardware platforms, a growing body of software, and increasing commercial interest. Sensor networks are seeing broader research and commercial deployments in military, scientific, and commercial applications including monitoring of biological habitats, agriculture, and industrial processes.

Sensor networks present challenges in three key areas. First, *energy consumption* is a common problem in sensor network design. Sensors are often battery operated and placed in remote locations, so any activity drains the sensor battery, bringing the node closer to death.[1] Second, how sensors *sense and interact* with the physical world is of great interest. Sensor networks

[1] Greg Pottie, personal communication.

focus on collaborative signal processing algorithms to exploit multiple, physically separate views on the environment. Finally, with tens, hundreds, or even thousands of sensor nodes, the network and applications as a whole must be *self-configuring*.

This chapter reviews each of these areas of sensor network design, beginning with common hardware platforms, then considering networking software and applications.

2 Hardware Platforms and Sensors

The wide availability of common hardware platforms, radios, and sensors has been an enabler for sensor networking in both the research and commercial communities.

Node hardware

A typical sensor node contains a general-purpose CPU and working memory, some kind of long-term stable storage such as flash memory or disk, and I/O capabilities to support sensors. Sensor nodes have evolved into two broad categories: *small devices* with 8-bit microcontrollers as CPUs, 10–100KB of working memory, and 100–1000KB of flash secondary storage; and *larger devices* with 32-bit CPUs and megabytes each of working memory and secondary storage.

Motes are representative of the smaller class of devices [22]. The current generation of Mica-2 motes uses an Atmega128 embedded processor running at about 4 MHz, providing 128KB flash memory for program code, 4KB working RAM, 8 channels of analog-to-digital converts, 48 digital I/O lines, a universal asynchronous receiver/transmitter (UART) and a serial peripheral interface (SPI). Motes have evolved significantly over the last several years; originally designed at Berkeley, they are now commercially available from several companies including Crossbow, Dust Networks, and Telos. Similar classes of embedded-controller-based devices are available from other academic and commercial institutions, with examples including the Nymph from the University of Colorado [1], and BTnodes from ETH Zurich [35].

The larger class of devices is exemplified by products such as the Stargate (designed by Intel, available from Crossbow Technologies) or the Cerfcube (from Intrinsyc). These devices are used in a variety of embedded applications. In the sensor network context, they are typically used as gateways to a collection of motes, or for applications that require heavier-duty signal processing. Each such device employs an X-Scale or ARM-based processor, has upwards of 64MB working memory, and 1GB of flash-based secondary storage. They support many connectivity options including USB and 802.11 wireless, and a 51-pin mote connector allowing use of a mote and its radio.

Power management is a concern in both classes of devices. Individual control of hardware components (CPU, storage, radio, sensors) is necessary for

power management. Many systems are battery powered. Energy harvesting is of growing importance, often via solar cells or vibration harvesting.

Sensors

Sensing technology has also kept pace with miniaturization of radios and processors, especially with the proliferation of microelectromechanical system (MEMS) sensors. Many such sensors have been incorporated into existing sensor node platforms.

Despite their diversity, the principle of operation behind most of these MEMS sensors is the same. They all rely on environmental factors inducing changes in the electrical properties of appropriately chosen materials. The sensors incorporate sensitive circuitry to detect changes in these electrical properties, and are calibrated to correctly measure the corresponding environmental phenomenon. For example, a temperature sensor relies on changes in the resistivity of certain materials with temperature. The choice of material ranges from metals to semiconductors and is dictated by the required sensing range and sensitivity. Similarly, a light sensor uses photoconductive materials whose electrical characteristics vary with the amount of light falling on them. Finally, accelerometers measure the voltage induced by structural deformations of piezo-electric materials; these deformations are caused by vibrations or by acceleration.

There is a very large industry devoted to manufacturing small MEMS sensors. This industry is segmented by application (e.g., companies such as Delphi cater to automotive manufacturers) and by sensor type (e.g., Silicon Designs focuses entirely on vibration sensors). However, only a handful of companies (examples include Ember and Millenial Net) focus on applications of wireless networked sensing.

3 Software and Protocols

Sensor networking has seen an enormous amount of research activity in the last five years, making it difficult to do justice to this large body of literature. Our exposition takes a systems approach, describing the components of an emerging general-purpose sensor networking infrastructure.

3.1 Networking

As the name implies, networking is a central component of sensor networks. Networking is important because it provides the glue that allows individual nodes to collaborate. In addition, the radio is a major consumer of energy in small sensor nodes, often 20–40% of the power draw when all components are on. Thus optimizing networking protocols can greatly extend the lifetime of the sensor network as a whole.

This section considers networking in sensor nets at the link layer, with media-access control (MAC) protocols, and at the network layer, with routing protocols. We also consider *topology control*, a service that can be part of either layer, or could be considered in between these two layers.

MAC protocols

Energy conservation is a key concern at the MAC protocols, and so before reviewing protocols we briefly describe MAC-related sources of energy consumption [50]. Packet *collisions* waste energy by forcing packets to be retransmitted, *idle listening* is the cost of actively listening for potential packets, *overhearing* is the cost of receiving packets intended for other destinations, and *control traffic* represents MAC-level maintenance overhead. Since many sensor networks are quiescent between sensor readings, idle listening can easily become the largest energy cost in a sensor net.

The IEEE 802.11 protocol (popularly known as "wi-fi") is contention-based MAC (carrier-sense, multiple-access or CSMA) now seeing wide commercial deployment. Intended for laptop computers, it provides good high-speed communication (up to 54Mb/s in some versions) for larger sensor network nodes. Unfortunately, the ad hoc mode of 802.11 that is required for peer-to-peer communications in a sensor network has very little support for energy conservation, and many sensor networks require bit rates less than 100kb/s, so the protocol is unsuitable for smaller and more power-constrained nodes.

Time-division multiple-access (TDMA) protocols were used in early sensor networks [43]. By scheduling media access they can largely avoid collisions, idle listening, and overhearing, thus greatly reducing energy consumption. Their disadvantage is that they often assume clustering, taxing the cluster head and making mobile operation more difficult.

Small sensor networks generally require relatively low-speed (20–40kb/s), simple protocols, precluding high-speed 802.11 and more complex TDMA protocols. Recent research has proposed 802.11-like MAC protocols designed to conserve energy by avoiding overhearing (PAMAS [42]) and idle listening (S-MAC [50, 51]). As an example protocol, S-MAC synchronizes most nodes into a sleep-schedule. Nodes regularly wake up, contend for the media if they have data to send, then either transmit data or go to sleep. By adjusting the sleep duration, duty cycles of 1–50% are possible to reduce the cost of idle listening. Adaptive listen [51] and future-Request-to-Send of T-MAC [46] extended these ideas to provide better throughput when there are multiple packets to send or when data travels over multiple hops.

IEEE 802.15.4 (also known as "ZigBee") is a recently standardized protocol targeted at sensor network and home automation applications. It includes an optional fixed duty cycle to avoid idle listening similar to research protocols such as we described previously. Although it is still too early to see how this protocol compares to current research, a standard protocol in this domain should spur commercial developments.

Network layer

As with MAC protocols, overhead is an important concern for sensor network routing protocols. Here the major source of overhead is control traffic: the number of routing update- or request-messages that are required. We next review routing protocols, both Internet protocol (IP)-based ad hoc networking and non-IP-based schemes.

The Internet Engineering Task Force (IETF) is standardizing *ad hoc* routing protocols for wireless, IP-based networks. Ad hoc routing protocols are usually grouped into *pro-active* protocols (for example, DSDV), which precompute routes to some or all destinations, and *reactive* protocols (for example, AODV and DSR), which compute routes to specific destinations only when prompted by traffic. The control traffic overhead of these protocols is proportional to the rate at which links change and, for reactive protocols, the rate traffic is sent to new destinations. Reactive protocols are a good match for very dynamic networks, such as those with many mobile nodes, since they only maintain routes to active destinations. Many sensor networks today have only stationary nodes. In these cases, pro-active protocols may be preferred because links change relatively infrequently, and such protocols are simpler and have no delay searching for a route when traffic is sent to a new destination.

In addition to IP-based routing protocols, geographic routing and directed diffusion are protocols more specific to sensor networks. Although originally proposed for wired networks, geographic routing protocols such as GPSR and similar protocols [3, 26] exploit the spatial nature of sensor networks and the spatial-dominated nature of radio propagation.

Directed diffusion combines a distance-vector-like, reactive routing protocol with an attribute-based routing scheme and an emphasis on processing data in the network [24]. Variants of directed diffusion provide several different routing mechanisms under the same interface [18]. Diffusion uses attribute-based routing instead of address-based or geographic routing. Diffusion combines attribute-based resource discovery with routing, and it allows applications to focus on the desired data rather than specific sensors. Diffusion suggests that processing data in the network is important for efficiency. Examples of in-network processing are duplicate suppression, data aggregation, and statistical filtering. When data is generated from multiple sensors, in-network processing can merge this data near those sites rather than sending all data out, thus greatly reducing energy consumption. (This approach has been adopted by several non-diffusion systems as well, for example, TinyDB, described below.)

3.2 Systems services

Beyond the lower-level networking primitives, a usable sensor networking system must provide several additional services. Many of these services, such

as operating systems, security, time synchronization, and resource discovery, are also found in traditional wired and wireless networks. However, some services, such as localization, are unique to sensor networks. In this subsection, we briefly discuss some of the services that sensor networks will implement, and describe challenges in the design of these services.

Operating systems and code development tools

The sensor networking community typically uses embedded (and, possibly, real-time) versions of existing operating systems such as Linux for the larger devices discussed above. These embedded versions provide largely the same programming support as their regular counterparts, but with additional device-level support for embedded controllers, flash memory, and other peripherals specific to these devices. As such, not much research has been required on new operating systems support for these larger devices.

By contrast, the smaller devices (such as the motes) have required novel directions in operating system design. One such direction has been the development of a POSIX-compliant multi-threaded OS for these small devices [1]. TinyOS [21], an operating system for the motes and widely used by many research groups as well as in some segments of industry, departs significantly from the traditional multi-threaded model of modern operating systems. Rather, TinyOS relies on the observation that most sensor networking applications will be *event-driven*: i.e., that applications will react to external sensed events. It is structured such that *components* (software modules that provide a distinct abstraction, either of a hardware device or of some software functionality such as a send-receive networking interface) can invoke *events* in other components. Typically, in response to a sensed event or some hardware interrupt, a chain of such component invocations can be used to process the event. In addition, TinyOS also provides two other abstractions: a *task*, which enables components to effectively use the idle processor for computations, and a *command*, which is invoked in order to get a component to perform an action (such as sending a network message or setting some device parameters).

Taken together, these abstractions allow a programmer to write an application as a component graph: nodes in this graph are components, and links signify event and command invocations from components or their associated tasks. In addition, the emphasis on event-driven programming rather than multi-threading avoids the memory cost of run-time stacks for each thread. This powerful functionality promotes code reuse, improves modularity (easy reuse of components), and minimizes object code size for memory-constrained devices.

Associated with TinyOS is a programming language called nesC, which contains language-level constructs for TinyOS's abstractions: components, tasks, events, and commands. This approach promotes compile-time program checking, as well as automated construction and management of the component graph. In addition, it provides support for parameterized, compile-time memory allocation to avoid the memory costs of dynamic memory allocation.

Finally, the community uses several simulation and emulation tools that enable code development, debugging, and system evaluation. One is *ns-2*, a general-purpose networking simulator, together with its wireless extensions. Ns-2 was widely used to develop and evaluate sensor networking routing protocols. More recently, sensor network specific simulators have seen increased use. Among these, TOSSIM [29] enables developers to simulate a network of motes and run actual application and protocol code on this network. Emstar [13] is a flexible programming environment for larger sensor nodes. As a simulation environment, it includes support for simulation of hybrid networks of large and small devices and several radio propagation and sensor models.

Node localization

Localization is the functionality by which nodes autonomously determine their *position* in two or three dimensions. This is a crucial service for sensor networks since location provides invaluable context in interpreting sensed data. In recent years, there has been significant work in localization for sensor networks and networks of embedded devices.

An important focus of the localization literature has been robust techniques for estimating distances between nodes (*ranging*). The networking community has focused on two classes of ranging techniques: radio frequency (RF)-based ranging and acoustic ranging. RF-based ranging, as exemplified by the SpotON [20] and Calamari [47] systems, is based on the premise that by measuring received signal strength a receiver can determine its distance to a transmitter. This presumes that RF propagation in an environment can be accurately characterized by a simple path loss model with known parameters. Using this technique, nodes can estimate distances to all neighbors within radio range. Range errors upwards of 10% of the nominal radio range have been reported in the literature [47], usually after a fairly involved calibration step that estimates the path loss parameters and adjusts for variations in transceiver characteristics. As an alternative to modeling radio propagation, the RADAR system has proposed building a database of receive signal strength as a function of location [2]. This approach requires surveying the environment to pre-compute the database and assumes propagation is relatively time invariant. A second class of ranging schemes is based on measuring the time-of-flight of an acoustic or ultrasound signal [14,39]. More precisely, these techniques measure the difference in arrival times of simultaneously transmitted radio and ultrasound signals, then estimate distances knowing the speed of sound. Some approaches in acoustic ranging use spread spectrum approaches for resilience to multipath effects, and employ techniques to correct for latencies induced by other system components [14]. Such techniques provide an order of magnitude better accuracy (1–2% error) than simple time-of-flight over distances of 3–6 meters.

Ranging is a component of a *localization system* of which there are, broadly speaking, two kinds: infrastructure-based and ad hoc. Systems in the former

class fix node positions by assuming the existence of some external infrastructure (typically beacons with known positions and with known or predetermined deployments). In this class, there has been extensive work on distributed position inference using specialized beacons [5], in-building localization systems that enable position for context-aware applications ([17], among others), and systems for estimating orientation of handheld devices [34].

Similarly, a large body of work has examined algorithms for ad hoc localization schemes. Perhaps the earliest pieces of work in the area of sensor network localization can be attributed to Bulusu et al. [5], Niculescu and Nath [32] and Savvides et al. [38,39]. Niculescu and Nath propose that nodes first estimate their distances to anchors using one of several techniques (DV-hop, DV-distance, and a Euclidean scheme), then fix their own position using these distances. Savvides et al. propose an N-hop multilateration scheme. That work also discusses a Kalman filtering-based position refinement phase to improve position estimates. In later work, they discuss the error characteristics and the dependence on network size and anchor density of their schemes [37]. Finally, Langendoen and Reijers [27] discuss a fairly detailed comparison of the above schemes in the face of ranging errors, different node densities, and anchor fractions.

Time synchronization

Sensor networks are predicated on the ability of sensor nodes to *collaboratively* detect events. Time synchronization is often a crucial requirement for collaborative detection—collaborating nodes may need to temporally correlate their sensor readings. The problem of node time synchronization has received extensive attention in the sensor networks literature. Much of this literature borrows heavily from early work on Internet time synchronization (NTP).

Networked time synchronization relies on time stamping a message at both the sender and receiver, and reconciling their clocks based on one or more such message exchanges. Between sending a message and receiving it there are several kinds of delays introduced: sender-side processing delay, message propagation delay, message transmission delay, and receiver-side processing delays. Techniques for time synchronization differ in how they estimate or eliminate various sources of delay.

For example, the simplest approach time stamps messages close to the radio hardware (thereby eliminating processing delays). Thus, a single message is sufficient to reconcile clocks if it is assumed that propagation delay is negligible. Two messages are sufficient to account for propagation delay as well [11]. Reference Broadcast Synchronization (RBS) avoids error introduced by variance in sender-side timing by comparing the same broadcast message received at the two *receivers*, and estimating receiver processing by averaging over several received packets [10]. As an aside, many of these techniques can be used to synchronize nodes after the occurrence of an event, an approach called *post-facto* synchronization [9].

Thus far, we have discussed how two nodes synchronize their clocks with each other. Some research has looked at synchronizing clocks network-wide to a reference [10, 11]. Most of these techniques rely on using one of the above methods to synchronize all the clocks to a reference clock hop by hop. Using such techniques, clock synchronization error increases linearly with network diameter. The existence of techniques that have better error characteristics is known theoretically, but no practical implementations of such techniques exist.

Resource discovery

Resource discovery is a problem growing out of the field of ubiquitous computing: When a device enters an area, how can it identify relevant local resources? In ubiquitous computing, relevant resources are network services such as printers and mail servers, to be used by people. Sun Microsystems' Jini includes resource discovery services for a local network, while web search engines such as Google can be thought of as Internet-wide resource discovery services.

Sensor networks today are typically more application specific and homogeneous, and so today there is little need to locate shared services. In many sensor networks today the only service not present at all nodes is a wide-area network connection; discovery of an Internet connection is easily integrated with routing. As sensor networks become more complex, we expect that individual nodes will become more heterogeneous. As cameras, additional storage, and other services are deployed, service discovery will become more important. In sensor networks today, directed diffusion combines resource discovery with routing [18]. Other protocols, such as ReOrg, support service heterogeneity (in its case, wall-powered nodes) integrated with the topology configuration protocol [8].

Databases and storage services

Individual sensor nodes in a sensor network produce many sensor readings. Nodes may also collaboratively detect events by exchanging these readings. We will collectively refer to readings and events as sensor data. An important systems challenge for sensor networking is the design of mechanisms for retrieving sensor data.

Protocols such as directed diffusion suggest data-centric communication as an architectural principle that governs the design of low-level mechanisms for accessing sensor data. At a higher level, a natural paradigm for accessing sensor data is to treat the sensor network as a distributed relational database. The Cougar [49] and TinyDB [31] systems allow users to specify sensor data of interest in a declarative fashion, using an SQL query of the form

```
SELECT AVG(temp)
  FROM sensors
  WHERE loc in (35,40,100,120) and light > 1525 lux
  SAMPLE PERIOD 35 seconds
```

Such a query allows the user to obtain the average temperature seen at all nodes observing a sufficiently high light intensity which are located within a specified region. These systems can be seen as providing a powerful, yet widely used, *programming* paradigm for sensor networks.

Both systems are implemented using data-centric communication primitives. For example, in TinyDB a query is flooded throughout the network, and nodes whose sensor data match the query respond. In principle, this is similar to interests and data messages in directed diffusion. Indeed, TinyDB can be implemented using directed diffusion.

While TinyDB and Cougar work well for continuous queries (those for which responses stream back continuously), the high overhead of flooding makes it unsuitable for one-shot queries. Researchers have focused on a class of systems that more efficiently support one-shot queries. These systems are built on an efficient rendezvous mechanism called *data-centric storage* [40]. In data-centric storage, a hash function is used to map a key associated with a data item to a geographic location. A geographic routing protocol called GPSR [26] is used to store the item at that location. A node wishing to retrieve items matching that key would use the same hash function and route a query to that node. Such a mechanism avoids flooding the query throughout the network. Depending on how the hash function is constructed, this mechanism can be used to construct a variety of storage structures that support sophisticated queries. These storage structures include distributed hash tables [36], distributed multi-dimensional indices [30], and storage hierarchies [12].

Remote programming

Reprogrammability is a key characteristic of software systems. While simple sensor networks may be configured "in the factory" and then discarded, sensors deployed for longer periods of time in remote locations motivate in situ reprogrammability to meet changing application requirements or just to fix bugs. Several flavors of reprogramming have been reconsidered by the community. *Retasking* usually refers to reconfiguring an application's parameters to match some pre-anticipated needs. *Scripting* and *virtual machines* represent the ability to reconfigure a sensor network at a high level, often by recombining pre-deployed lower-level components. Finally, true *reprogramming* is reserved for replacing the complete operating image of the sensor node.

Retasking has been explored in several environments. One example of retasking is directed diffusion [24] and filters [19], where application-specific attributes can be used to tune a fielded application. Another widely used example is TinyDB [31], where new queries are downloaded into the network. In

both of these cases, run-time information is distributed through the network to reconfigure pre-deployed filters or database operators.

A more generic facility is possible with scripting or virtual machines. In SensorWare [4], pre-configured components are reconfigured on-the-fly by commands distributed over the network. Scripting is distinguished from retasking because the configuration information is provided by a script in a high-level language (Tcl, in the case of SensorWare), allowing more sophisticated reconfiguration than is possible with the static data structures of simple retasking. Virtual machines were popularized with Java; Maté is an example of virtual machines applied to sensor networks [28]. While traditional virtual machines provide a primitive, low-level instruction set, Maté emphasizes very high-level, application-specific instructions to maximize code density.

Reprogramming the complete sensor node is a delicate process in which a software image is transferred over one or more hops, verified, and then the sensor node is carefully rebooted to run the new code. Two recent systems have described mote-level reprogrammability: Hui and Culler's system [23] and MOAP [44]. Both carefully segment and transfer a relatively large (multi-kilobyte) software image; Hui's system emphasizes rapidly propagating the image throughout the entire network with pipelining, while MOAP strives to transfer a complete image to nearby neighbors before forwarding data further.

Many of the above approaches assume the entire sensor network is to be reprogrammed. Since messages in diffusion are addressed to nodes identified by particular attributes it is easy to retask part of a sensor network. SensorWare has also used scripts to reprogram parts of a network. In principle, similar techniques could be applied to the other approaches.

Security

Security issues have not received as much attention as some of the other research areas in sensor networks. This is understandable, since often interesting security research is spurred by vulnerabilities learned from large-scale deployments, of which there are relatively few. There is, of course, a general acceptance that security is of paramount importance in sensor networks. They are vulnerable to a wide variety of denial of service attacks at all levels, ranging from the physical to the application layer [48].

Existing sensor network security research has mostly focused on adapting security mechanisms to the computational and messaging constraints imposed by tiny sensor devices [25, 33]. This line of research attempts to implement encryption and message authentication mechanisms by relying on shared symmetric or group keys augmented with message counters. Some of these mechanisms have been implemented on the motes in order to provide secure communication at the link layer.

3.3 Application primitives

Sensor networks will, in general, be used to sense phenomena of different kinds. Broadly speaking, phenomena may be of two kinds: *diffuse* phenomena like fires, clouds, contaminants, etc., and *point* phenomena like animals, tanks, and other targets. Generic techniques for sensing these kinds of phenomena might form useful application-level primitives which ease the task of developing new applications. Some sensor networks research has focused on primitives for these two kinds of phenomena.

Diffuse phenomena are distinguished by their spatial extent, which is generally larger than the average inter-sensor spacing. Thus, an important primitive for such phenomena is one that detects and tracks the *boundary* of the phenomenon. Not much research attention has been bestowed on this class of problems, aside from isolated pieces of work that have discussed a technique to compute the approximate boundary of a phenomenon along a hierarchical structure, and techniques to robustly, yet locally, estimate whether a node lies on the boundary of a phenomenon or not.

Rather more attention has been devoted to application-level primitives for point phenomena. One can decompose the problem of sensing such phenomena into two smaller problems: *target localization*, which determines *where* the point phenomenon or target is, and *tracking*, which updates the path of the target as it moves. Both of these problems have been examined in the sensor network context.

For target localization, a simple technique would be to use the "closest point of approach", i.e., to say that the location of the target is the location of the sensor which detects the target with the highest intensity. Variants of this algorithm might pinpoint the target as being located at the weighted centroid of all sensors that sense the target. The accuracy of such techniques depends heavily on deployment density. A more sophisticated approach, and one that has been studied fairly extensively in the signal processing literature, relies on *triangulating* the target position based on the observed delay differences in the received signal at a cluster of sensors within the network. This technique has been shown to work quite well [7].

Having localized the target, the next challenge is to track the target as it moves through the sensor field. A simple representation of the target's track is the sequence of its locations over time, and this may be computed by sending the target's location periodically to a base station. This approach can incur significant communication cost, so more sophisticated techniques rely on handing off the track to sensors along the target's path. Furthermore, it is possible to be more energy efficient by *waking up* sensors in advance of a target's arrival. However, this requires techniques that can predict the target's track. A body of work has focused on *information-directed* approaches to solve this problem [52]. These approaches maintain a continuously updated belief state about target location that allows them to probabilistically determine to which sensor the target's track should be handed off.

4 Applications

The sensor network community is investigating several disciplines in which sensor networks might be applicable for various purposes. The following paragraphs discuss these potential applications briefly, sketching applications in the military, the sciences and environmental monitoring, and civil and industrial areas. For many of these applications, sensor networks will enable in situ sensing at unprecedented spatial scales.

Military applications

Military applications supported much early work in sensor networks. Securing an area to detect intruders and monitoring vehicle traffic on a road or in open terrain were a focus of the DARPA SensIT program. More recently researchers have demonstrated a sensor-network-based sniper localization system [41].

Environmental monitoring applications

Many current applications for sensor networks are in areas of biology and life sciences, where a common theme is the ability of sensors to take observations in much more detail and for much longer than is possible today. We briefly evaluate habitat monitoring, marine microorganism monitoring, contaminant transport, and precision farming.

Habitat monitoring has been the focus of great interest in the sensor network community [45]. Examples include micro-climate monitoring at James Reserve, and nest monitoring at Great Duck Island (see [45] for details and additional references). These applications provide an ideal testing ground for sensor networks because they require fairly simple monitoring (light, temperature, sound, perhaps presence or absence of an animal) at tens of stations. This level of monitoring is not possible without sensor networks because human observations would be too invasive to the environment and centralized or wired monitors cannot span the physical area.

Marine biologists envision using sensor networks to obtain data at fine spatial scales (a few meters to tens of meters). There is a need for such data in their application domain, and current instrumentation technology is inadequate or too expensive to fulfill this need. The time evolution of red tides (rapidly formed colonies of algae that are harmful to fish and birds) is poorly understood, and appears to be triggered by small scale temperature, light, and nutrient variations. Sensor networks can be deployed at this scale, and have been used in a laboratory setting to gather data. An interesting twist is the addition of limited actuation to such networks, where the sensors may move (e.g., in a small boat) a little in order to obtain better quality data or to vary spatial coverage.

A similar use is envisioned by environmental engineers, who see sensor networks helping them build accurate models of contaminant seepage in soil. Data at fine spatial scales can be used to more precisely model contaminant

flow [16] and thus predict contamination of scarce groundwater resources. In the longer term, such networks can be used for monitoring the compliance of industries to regulations that govern the release of contaminants into the soil. A closely related area is precision farming, where detailed monitoring enabled by dense sensor deployment could allow more effective use of fertilizers.

Civil and commercial applications

Finally, there is growing interest in sensor networks in civil engineering and industrial applications.

Seismologists envision using sensor networks to understand the propagation of earthquakes at fine spatial scales. This propagation is critically affected by soil conditions, and can impact how much earthquakes affect buildings and other structures. A related application is structural monitoring [6]: sensor networks can be used to measure the response of a building to vibrations, and the variation of these responses over time can be used to detect and localize damage in a variety of structures (buildings, bridges, ships).

Transportation networks are an important economic part of all cities, and it is not surprising that there is a fairly large investment in traditional fixed sensors and centralized traffic monitoring systems. Researchers are exploring how sensor networks can augment this infrastructure in two different ways. Rapidly deployable sensor networks for traffic monitoring may be useful to temporarily collect data for development or pollution-related traffic studies in areas that do not warrant long-term monitoring [15]. More radically, several research groups have proposed a future where each car has its own sensors that can communicate with nearby cars, avoiding centralized management and enabling new applications.

Finally, there is growing interest in industrial applications of sensor networks to closely monitor manufacturing and safety conditions. Although these applications are just now emerging, promising areas include industrial monitoring in the oil industry (Ember), environmental monitoring in semiconductor processing facilities (Intel), and even monitoring of art in museums (Sensicast).

5 Conclusions

This chapter has surveyed embedded sensor networks. With recent hardware advances for small, inexpensive, networked sensors, a growing body of software components to link them together into a whole, and applications in many areas, embedded sensor networks are an active and growing area of embedded computing.

References

1. H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System support for MultimodAl NeTworks of In-situ Sensors. In *Proceedings of the 2nd ACM Workshop on Sensor Networks and Applications*, pages 50–59, San Diego, CA, USA, Sept. 2003. ACM.
2. P. Bahl and V. N. Padmanabhan. RADAR: An in-building RF-based user location and tracking system. In *Proceedings of the IEEE Infocom*, pages 775–784, Tel Aviv, Israel, Mar. 2000. IEEE.
3. P. Bose, P. Morin, I. Stojmenovic, and J. Urrutia. Routing with guaranteed delivery in ad hoc wireless networks. In *Proceedings of the Third ACM International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications (Dial M)*, pages 48–55, Seattle, WA, USA, Aug. 1999. ACM.
4. A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the ACM/Usenix International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pages 187–200, San Francisco, CA, USA, May 2003. ACM.
5. N. Bulusu, J. Heidemann, and D. Estrin. GPS-less low cost outdoor localization for very small devices. *IEEE Personal Communications Magazine*, 7(5):28–34, Oct. 2000.
6. J. Caffrey, R. Govindan, E. Johnson, B. Krishnamachari, S. Masri, and G. Sukhatme. Networked sensing for structural health monitoring. In *Proceedings of the Fourth International Workshop on Structural Monitoring and Control*, June 2004.
7. J. Chen, L. Yip, J. Elson, H. Wang, D. M. R. Hudson, K. Yao, and D. Estrin. Coherent acoustic array processing and localization in wireless sensor networks. *Proceedings of the IEEE*, 2003.
8. W. S. Conner, J. Chhabra, M. Yarvis, and L. Krishnamurthy. Experimental evaluation of synchronization and topology control for in-building sensor network applications. In *Proceedings of the Second ACM Workshop on Sensor Networks and Applications*, pages 38–49, San Diego, CA, USA, Sept. 2003. ACM.
9. J. Elson and D. Estrin. Time synchronization for wireless sensor networks. In *Proceedings of the 15th IEEE International Parallel and Distributed Processing Symposium*, pages 1965–1970, San Francisco, CA, USA, Apr. 2001. IEEE.
10. J. Elson, L. Girod, and D. Estrin. Fine-grained network time synchronization using reference broadcasts. In *Proceedings of the Fifth Symposium on Operating Systems Design and Implementation (OSDI 2002)*, Boston, MA, December 2002.
11. S. Ganeriwal, R. Kumar, and M. B. Srivastava. Timing-sync protocol for sensor networks. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, pages 138–149. ACM Press, 2003.
12. D. Ganesan, D. Estrin, and J. Heidemann. DIMENSIONS: Why do we need a new data handling architecture for sensor networks? In *Proceedings of the ACM Workshop on Hot Topics in Networks*, pages 143–148, Princeton, NJ, USA, Oct. 2002. ACM.
13. L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. Emstar: A software environment for developing and deploying wireless sensor networks. In *Proceedings of the USENIX Conference Proceedings*, pages 283–296, Boston, MA, USA, June 2004. USENIX.

14. L. Girod and D. Estrin. Robust range estimation using acoustic and multimodal sensing. In *Proc. IEEE International Conference on Intelligent Robots and Systems*, Maui, HI, USA, Oct. 2001. IEEE.
15. G. Giuliano and J. Heidemann. Rapidly deployable sensors for vehicle counting and classification. `http://www.isi.edu/ilense/metrans/`, 2004. Work in progress.
16. T. Harmon. Networked Sensing in Support of Real-time Parameter Estimation. Association of Environmental Engineering and Science Professors, 2003.
17. A. Harter, A. Hopper, P. Steggles, A. Ward, and P. Webster. The anatomy of a context-aware application. In *Proceedings of the ACM International Conference on Mobile Computing and Networking*, pages 59–67, Seattle, WA, USA, Aug. 1999. ACM.
18. J. Heidemann, F. Silva, and D. Estrin. Matching data dissemination algorithms to application requirements. In *Proceedings of the ACM SenSys Conference*, pages 218–229, Los Angeles, CA, USA, Nov. 2003. ACM.
19. J. Heidemann, F. Silva, Y. Yu, D. Estrin, and P. Haldar. Diffusion filters as a flexible architecture for event notification in wireless sensor networks. Technical Report ISI-TR-556, USC/Information Sciences Institute, Apr. 2002.
20. J. Hightower, C. Vakili, G. Borriello, and R. Want. Design and Calibration of the SpotON Ad-Hoc Location Sensing System, 2001. Unpublished manuscript.
21. J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System archtecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.
22. J. L. Hill and D. E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, Nov/Dec 2002.
23. J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd ACM SenSys Conference*, pages 81–94, Baltimore, MD, USA, Nov. 2004. ACM.
24. C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: A scalable and robust communication paradigm for sensor networks. In *Proceedings of the ACM International Conference on Mobile Computing and Networking*, pages 56–67, Boston, MA, USA, Aug. 2000. ACM.
25. C. Karlof, N. Sastry, and D. Wagner. TinySec: Link-Layer Encryption for Tiny Devices. In *Proceedings of the 2nd ACM SenSys Conference*, pages 162–175, 2004.
26. B. Karp and H. T. Kung. GPSR: Greedy perimeter stateless routing for wireless networks. In *Proceedings of the ACM International Conference on Mobile Computing and Networking*, pages 243–254, Boston, MA, USA, Aug. 2000. ACM.
27. K. Langendoen and N. Reijers. Distributed Localization in Wireless Sensor Networks: A Quantitative Comparison. Technical Report PDS-2002-003, Technical University, Delft, November 2002.
28. P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, San Jose, CA, USA, Oct. 2002. ACM.
29. P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: accurate and scalable simulation of entire tinyOS applications. In *Proceedings of the First International Conference on Embedded Networked Sensor Systems*, pages 126–137, Los Angeles, CA, USA, 2003. ACM Press.

30. X. Li, Y. J. Kim, R. Govindan, and W. Hong. Multi-dimensional range queries in sensor networks. In *Proceedings of the ACM SenSys Conference*, pages 63–75, Los Angeles, CA, USA, Nov. 2003.
31. S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: Tiny AGgregate queries in ad-hoc sensor networks. In *Proceedings of the Usenix Symposium on Operating Systems Design and Implementation*, pages 131–146, Boston, MA, USA, Dec. 2002. USENIX.
32. D. Niculescu and B. Nath. Ad-hoc positioning system. In *Proceedings of IEEE Globecom*, 2001.
33. A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. SPINS: Security Protocols for Sensor Networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 189–199. ACM Press, 2001.
34. N. B. Priyantha, A. K. L. Miu, H. Balakrishnan, and S. Teller. The cricket compass for context-aware mobile applications. In *Proc. of Sixth ACM International Conference on Mobile Computing and Networking (MOBICOM)*, Rome, Italy, July 2001.
35. T. S.-I. Project. Smart-its home page. `http://www.smart-its.org/`, 2003.
36. S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. GHT: A geographic hash table for data-centric storage. In *Proceedings of the ACM Workshop on Sensor Networks and Applications*, pages 78–87, Atlanta, GA, USA, Sept. 2002. ACM.
37. A. Savvides, W. Garber, S. Adlakha, R. Moses, and M. Srivastava. On the error characteristics of multihop node localization in wireless sensor networks. In *Proceedings of First International Workshop on Information Processing in Sensor Networks*, 2003.
38. A. Savvides, C.-C. Han, and M. Srivastava. Dynamic fine-grained localization in ad-hoc networks of sensors. In *Proc. of Seventh ACM International Conference on Mobile Computing and Networking (MOBICOM)*, pages 166–179, Rome, Italy, July 2001. ACM.
39. A. Savvides, H. Park, and M. Srivastava. The bits and flops of the N-hop multilateration primitive for node localization problems. In *Proceedings of the First International Workshop for Wireless Sensor Networks and Applications (WSNA)*, 2002.
40. S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. Data-centric storage in sensornets. In *Proc. ACM SIGCOMM Workshop on Hot Topics In Networks*, pages 137–144, Princeton, NJ, 2002.
41. G. Simon, A. Ledeczi, and M. Maroti. Sensor network-based countersniper system. In *Proceedings of the 2nd ACM SenSys Conference*, pages 1–12, Baltimore, MD, USA, Nov. 2004. ACM.
42. S. Singh and C. Raghavendra. PAMAS: Power aware multi-access protocol with signalling for ad hoc networks. *ACM Computer Communication Review*, 28(3):5–26, July 1998.
43. K. Sohrabi, J. Gao, V. Ailawadhi, and G. Pottie. A self-organizing sensor network. In *Proceedings of the 37th Allerton Conference on Communication, Control, and Computing*, Monticello, IL, USA, Sept. 1999.
44. T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Sensing, Nov. 2003.

45. R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin. Application driven systems research: Habitat monitoring with sensor networks. *Communications of the ACM*, 47(6):34–40, June 2004.
46. T. van Dam and K. Langendoen. An adaptive energy-efficient mac protocol for wireless sensor networks. In *Proceedings of the First ACM SenSys Conference*, pages 171–180, Los Angeles, CA, USA, Nov. 2003. ACM.
47. K. Whitehouse and D. Culler. Calibration as a parameter estimation problem in sensor network. In *Proceedings of the ACM Workshop on Sensor Networks and Applications*, Atlanta, GA, 2002.
48. A. D. Wood and J. A. Stankovic. Denial of service in sensor networks. *IEEE Computer*, 35(10):54–62, Oct. 2002.
49. Y. Yao and J. Gehrke. The Cougar approach to in-network query processing in sensor networks. In *SIGMOD Record*, September 2002.
50. W. Ye, J. Heidemann, and D. Estrin. An energy-efficient MAC protocol for wireless sensor networks. In *Proceedings of the IEEE Infocom*, pages 1567–1576, New York, NY, USA, June 2002. USC/Information Sciences Institute, IEEE.
51. W. Ye, J. Heidemann, and D. Estrin. Medium access control with coordinated, adaptive sleeping for wireless sensor networks. *ACM/IEEE Transactions on Networking*, 12(3):493–506, June 2004. A preprint of this paper was available as ISI-TR-2003-567.
52. F. Zhao, J. Shin, and J. Reich. Information-driven dynamic sensor collaboration for tracking applications. *IEEE Signal Processing Magazine*, 19(2):61–72, Mar. 2002.

Part VI

Applications

Vehicle Applications of Controller Area Network

Karl Henrik Johansson,[1,*] Martin Törngren,[2,†] and Lars Nielsen[3]

[1] Department of Signals, Sensors and Systems, Royal Institute of Technology, Stockholm, Sweden, kallej@s3.kth.se
[2] Department of Machine Design, Royal Institute of Technology, Stockholm, Sweden, martin@damek.kth.se
[3] Department of Electrical Engineering, Linköping University, Sweden, lars@isy.liu.se

1 Introduction

The Controller Area Network (CAN) is a serial bus communications protocol developed by Bosch in the early 1980s. It defines a standard for efficient and reliable communication between sensor, actuator, controller, and other nodes in real-time applications. CAN is the de facto standard in a large variety of networked embedded control systems. The early CAN development was mainly supported by the vehicle industry: CAN is found in a variety of passenger cars, trucks, boats, spacecraft, and other types of vehicles. The protocol is also widely used today in industrial automation and other areas of networked embedded control, with applications in diverse products such as production machinery, medical equipment, building automation, weaving machines, and wheelchairs.

In the automotive industry, embedded control has grown from stand-alone systems to highly integrated and networked control systems [7, 11]. By networking electro-mechanical subsystems, it becomes possible to modularize functionalities and hardware, which facilitates reuse and adds capabilities. Fig. 1 shows an example of an electronic control unit (ECU) mounted on a diesel engine of a Scania truck. The ECU handles the control of engine, turbo, fan, etc. but also the CAN communication. Combining networks and mechatronic modules makes it possible to reduce both the cabling and the number

*The work of K. H. Johansson was partially supported by the European Commission through the ARTIST2 Network of Excellence on Embedded Systems Design, by the Swedish Research Council, and by the Swedish Foundation for Strategic Research through an Individual Grant for the Advancement of Research Leaders.

†The work of M. Törngren was partially supported by the European Commission through ARTIST2 and by the Swedish Foundation for Strategic Research through the project SAVE.

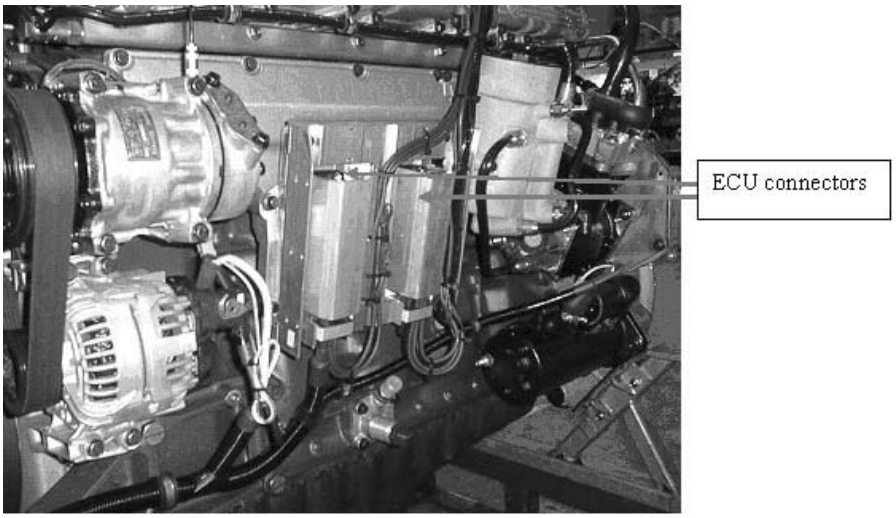


Fig. 1. An ECU mounted directly on a diesel engine of a Scania truck. The arrows indicate the ECU connectors, which are interfaces to the CAN. (Courtesy of Scania AB.)

of connectors, which facilitates production and increases reliability. Introducing networks in vehicles also makes it possible to more efficiently carry out diagnostics and to coordinate the operation of the separate subsystems.

The CAN protocol standardizes the physical and data link layers, which are the two lowest layers of the open systems interconnection (OSI) communication model (see Fig. 2). For most systems, higher-layer protocols are needed to enable efficient development and operation. Such protocols are needed for defining how the CAN protocol should be used in applications, for example, how to refer to the configuration of identifiers with respect to application messages, how to package application messages into frames, and how to deal with start-up and fault handling. Note that in many cases only a few of the OSI layers are required. Note also that real-time issues and redundancy management are not covered by the OSI model. The adoption of CAN in a variety of application fields has led to the development of several higher-layer protocols, including SAE J1939, CANopen, DeviceNet, and CANKingdom. Their characteristics reflect differences in requirements and traditions of application areas. An example is the adoption of certain communication models, such as either the client-server model or the distributed data-flow model [13].

The progress and success of CAN are due to a number of factors. The evolution of microelectronics paved the way for introducing distributed control in vehicles. In the early 1980s there was, however, a lack of low-cost and standardized protocols suitable for real-time control systems. Therefore, in 1983 Kiencke started the development of a new serial bus system at Bosch,

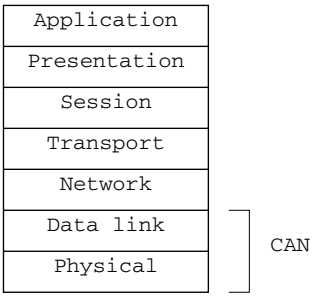


Fig. 2. The CAN protocol defines the lowest two layers of the OSI model. There exist several CAN-based higher-layer protocols that are standardized. The user choice depends on the application.

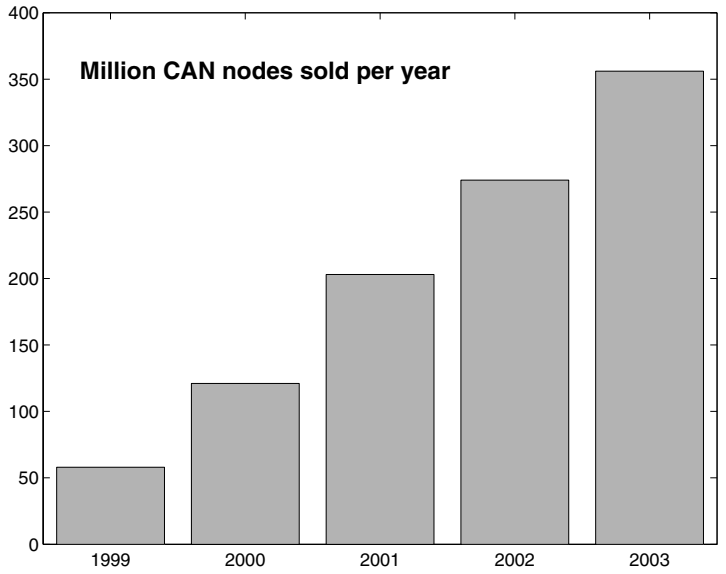


Fig. 3. The number of CAN nodes sold per year is currently about 400 million. (Data from the association CAN in Automation [3].)

which was presented as CAN in 1986 at the SAE congress in Detroit [8]. The CAN protocol was internationally standardized in 1993 as ISO 11898-1. The development of CAN was mainly motivated by the need for new functionality, but it also reduced the need for wiring. The use of CAN in the automotive industry has caused mass production of CAN controllers. Today, CAN controllers are integrated on many microcontrollers and available at a low cost. Fig. 3 shows the number of CAN nodes that were sold during 1999–2003.

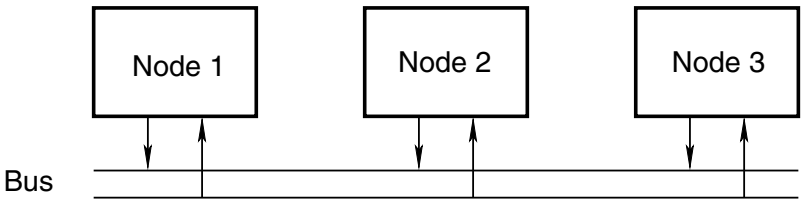


Fig. 4. Three nodes connected through a CAN bus

The purpose of this chapter is to give an introduction to CAN and some of its vehicle applications. The outline is as follows. Section 2 describes the CAN protocol, including its message formats and error handling. The section is concluded by a brief history of CAN. Examples of vehicle application architectures based on CAN are given in Section 3. A few specific control loops closed over CAN buses are discussed in Section 4. The paper is concluded with some perspectives in Section 5, where current research issues such as x-by-wire and standardized software architectures are considered. The examples are described in more detail in [14]. A detailed description of CAN is given in the textbook [6]. Another good resource for further information is the homepage of the organization CAN-in-Automation (CiA) [3]. The use of CAN as a basis for distributed control systems is discussed in [13].

2 Controller Area Network

The Controller Area Network (CAN) is a serial communications protocol suited for networking sensors, actuators, and other nodes in real-time systems. In this section, we first give a general description of CAN including its message formats, principle of bus arbitration, and error-handling mechanisms. Extensions of CAN, such as application-oriented higher-layer protocols and time-triggered CAN, are described, followed by a brief history of CAN.

2.1 Description

A CAN bus with three nodes is depicted in Fig. 4. The CAN specification [4] defines the protocols for the physical and the data link layers, which enable the communication between the network nodes. The application process of a node, e.g., a temperature sensor, decides when it should request the transmission of a message frame. The frame consists of a data field and overhead, such as identifier and control fields. Since the application processes in general are asynchronous, the bus has a mechanism for resolving conflicts. For CAN, it is based on a non-destructive arbitration process. The CAN protocol therefore belongs to the class of protocols denoted as carrier sense multiple access/collision avoidance (CSMA/CA), which means that the protocol listens

Fig. 5. CAN message frame

to the network in order to avoid collisions. CSMA/CD protocols like Ethernet have instead a mechanism to deal with collisions once they are detected. CAN also includes various methods for error detection and error handling. The communication rate of a network based on CAN depends on the physical distances between the nodes. If the distance is less than 40 m, the rate can be up to 1 Mbps.

Message formats

CAN distinguishes four message formats: data, remote, error, and overload frames. Here we limit the discussion to the data frame, shown in Fig. 5. A data frame begins with the start-of-frame (SOF) bit. It is followed by an eleven-bit identifier and the remote transmission request (RTR) bit. The identifier and the RTR bit form the arbitration field. The control field consists of six bits and indicates how many bytes of data follow in the data field. The data field can be zero to eight bytes. The data field is followed by the cyclic redundancy checksum (CRC) field, which enables the receiver to check if the received bit sequence was corrupted. The two-bit acknowledgment (ACK) field is used by the transmitter to receive an acknowledgment of a valid frame from any receiver. The end of a message frame is signaled through a seven-bit end-of-frame (EOF). There is also an extended data frame with a twenty-nine-bit identifier (instead of eleven bits).

Arbitration

Arbitration is the mechanism that handles bus access conflicts. Whenever the CAN bus is free, any unit can start to transmit a message. Possible conflicts, due to more than one unit starting to transmit simultaneously, are resolved by bit-wise arbitration using the identifier of each unit. During the arbitration phase, each transmitting unit transmits its identifier and compares it with the level monitored on the bus. If these levels are equal, the unit continues to transmit. If the unit detects a dominant level on the bus, while it was trying to transmit a recessive level, then it quits transmitting (and becomes a receiver). The arbitration phase is performed over the whole arbitration field. When it is over, there is only one transmitter left on the bus.

The arbitration is illustrated by the following example with three nodes (see Fig. 6). Let the recessive level correspond to “1” and the dominant level to “0”, and suppose the three nodes have identifiers I_i, $i = 1, 2, 3$, equal to

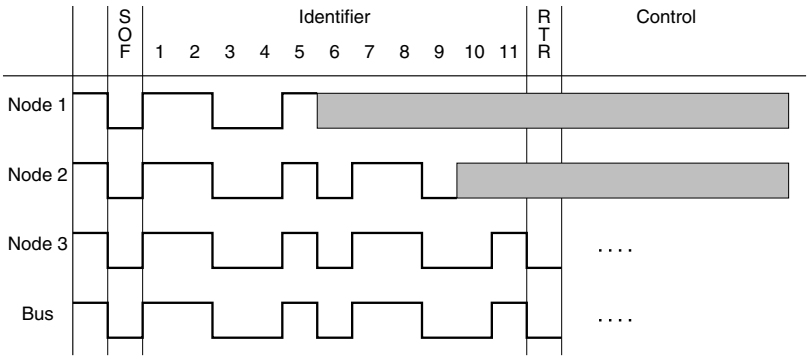


Fig. 6. Example illustrating CAN arbitration when three nodes start transmitting their SOF bits simultaneously. Nodes 1 and 2 stop transmitting as soon as they transmit one (recessive level), and Node 3 is transmitting zero (dominant level). At these instances, Nodes 1 and 2 enter the receiver mode, indicated in grey. When the identifier has been transmitted, the bus belongs to Node 3 which thus continues transmitting its control field, data field, etc.

$$I_1 = 11001101010, \quad I_2 = 11001011011, \quad I_3 = 11001011001.$$

If the nodes start transmitting simultaneously, the arbitration process illustrated in the figure takes place. First all three nodes send their SOF bit. Then, they start transmitting their identifiers and all of them continue as long as they are equal. The sixth bit of I_1 is at the recessive level, while the corresponding bits are at the dominant level of I_2 and I_3. Therefore, Node 1 stops transmitting immediately and continues only listening to the bus. This listening phase is indicated in Fig. 6 with the grey field. Since the tenth bit of I_2 is at the recessive level while it is dominant for I_3, Node 3 is the transmitter that has access to the bus after the arbitration phase and thus continues with the transmission of the control and data fields, etc.

There is no notion of message destination addresses in CAN. Instead each node picks up all traffic on the bus. Hence, every node needs to filter out the interesting messages on the bus. The arbitration mechanism of CAN is an effective way to resolve bus conflicts. Note that a minimum amount of address data is transmitted and that no extra bus control information has to be transmitted. A consequence of the arbitration mechanism, however, is that units with low priority may experience large latency if high-priority units are very active.

Error handling

Error detection and error handling are important for the performance of CAN. Because of complementary error detection mechanisms, the probability of having an undetected error is very small. Error detection is done in five different

ways in CAN: bit monitoring and bit stuffing, as well as frame check, ACK check, and CRC. Bit monitoring simply means that each transmitter monitors the bus level, and signals a bit error if the level does not agree with the transmitted signal. (Bit monitoring is not done during the arbitration phase.) After having transmitted five identical bits, a node will always transmit the opposite bit. This extra bit is neglected by the receiver. The procedure is called bit stuffing, and it can be used to detect errors. The frame check consists of checking that the fixed bits of the frame have the values they are supposed to have, e.g., EOF consists of seven recessive bits. During the ACK in the message frame, all receivers are supposed to send a dominant level. If the transmitter, which transmits a recessive level, does not detect the dominant level, then an error is signaled by the ACK check mechanism. Finally, the CRC is that every receiver calculates a checksum based on the message and compares it with the CRC field of the message.

Every receiver node obviously tries to detect errors within each message. If an error is detected, it leads to an immediate and automatic retransmission of the incorrect message. In comparison to other network protocols, this mechanism leads to a high data integrity and a short error recovery time. CAN thus provides elaborate procedures for error handling, including retransmission and reinitialization. The procedures have to be studied carefully for each application to ensure that the automated error handling is in line with the system requirements.

2.2 Protocol extensions

CAN provides the basic functionality described above. In many situations, it is desirable to use standardized protocols that define the communication layers on top of the CAN. Such higher-layer protocols are described below together with CAN gateways and the time-triggered extension of CAN denoted TTCAN, which allows periodic access to the communication bus with a high degree of certainty.

Higher-layer protocols

The CAN protocol defines the lowest two layers of the OSI model in Fig. 2. In order to use CAN, protocols are needed to define the other layers. Fieldbus protocols usually do not define the session and presentation layers, since they are not needed in these applications. The users may either decide to define their own software for handling the higher layers, or they may use a standardized protocol. Existing higher-layer protocols are often tuned to a certain application domain. Examples of such protocols include SAE J1939, CANopen, and DeviceNet. It is only SAE J1939 that is specially developed for vehicle applications. Recently, attempts have been made to interface CAN and Ethernet, which is the dominant technology for local area networks and widely applied for connecting to the Internet.

SAE J1939 is a protocol that defines the higher-layer communication control. It was developed by the American Society of Automotive Engineers (SAE) and is thus targeted to the automotive industry. The advantage of having a standard is considerable, since it enables independent development of the individual networked components, which also allows vehicle manufacturers to use components from different suppliers. SAE J1939 specifies, e.g., how to read and write data, but also how to calibrate certain subsystems. The data rate of SAE J1939 is about 250 kbps, which gives up to about 1850 messages per second [6]. Applications of SAE J1939 include truck-and-trailer communication, vehicles in agriculture and forestry, as well as navigation systems in marine applications.

CANopen is a standardized application defined on top of CAN and widely used in Europe for the application of CAN in distributed industrial automation. It is a standard of the organization CAN in Automation (CiA) [3]. CANopen specifies communication profiles and device profiles, which enable an application-independent use of CAN. The communication profile defines the underlying communication mechanism. Device profiles exist for the most common devices in industrial automation, such as digital and analog I/O components, encoders, and controllers. The device can be configured through CANopen independent of its manufacturer. CANopen distinguishes real-time data exchange and less critical data exchange. It provides standardized communication objects for real-time data, configuration data, network management data, and certain special functions (e.g., time stamp and synchronization messages).

DeviceNet is another standardized application defined on top of CAN for distributed industrial automation. It is mainly used in the U.S.A. and Asia and was originally developed by Rockwell Automation. DeviceNet, ControlNet, and transmission control protocol/Internet protocol (TCP/IP) are open network technologies that share upper layers of the communication protocol, but are based on lower layers: DeviceNet is built on CAN, ControlNet on a token-passing bus protocol, and TCP/IP on Ethernet.

CANKingdom is a high-layer protocol used for motion control systems. It is also used in the maritime industry, as described in a boat example in Section 3. CANKingdom allows the changing of network behavior at any time, including while the system is running. For example, CANKingdom allows the system troubleshooter to turn off individual nodes. The CAN node identifiers and the triggering conditions for sending messages can be changed while the system is running. One instance when real-time network reconfiguration is used is during failure conditions. An example is a loss of a radio link ECU in a maritime application. The network monitor, also known as the King, in that case first shuts off the radio node to keep it from sending any more commands, and then tells the appropriate nodes to get data from the King. This operation eliminates the problem of a node receiving two simultaneous but conflicting commands. It also eliminates the problem of two nodes sending the same CAN id.

The high-level protocols described above have been developed with different applications and traditions in mind, which is reflected, for example, in their support for real-time control. Although SAE J1939 is used for implementing control algorithms, it does not provide explicit support for time-constrained messaging. In contrast, such functionalities are provided by CANKingdom and CANopen, which handle explicit support for inter-node synchronization. CANKingdom and CANopen allow static and dynamic configuration of the network, whereas SAE J1939 provides little flexibility.

CAN gateways

Gateways and bridges enable CAN-based networks to be linked together or linked to networks with other protocols. A gateway between a CAN and another communication network maps the protocols of the individual networks. There exist many different types of CAN gateways, e.g., CAN-RS232 and CAN-TCP/IP gateways. The latter can provide remote access to a CAN through the Internet, which allows worldwide monitoring and maintenance. The networks connected through a gateway or a bridge are disconnected in terms of their real-time behavior, so obviously the timing and performance of the complex inter-connected network can be hard to predict even if the individual networks are predictable.

Ethernet (or rather Ethernet/IP) is quite a different communication protocol compared to CAN, but is still of growing importance in industrial automation either in constellations with CAN or on its own. Traditionally, Ethernet is used in office automation and multimedia applications, while CAN dominates in vehicles and in certain industrial automation systems. The strength of Ethernet is the ability to quickly move large amounts of data over long distances and that the number of nodes in the network can be large. CAN, on the other hand, is optimized for transmitting small messages over relatively short distances. A drawback with a network based on the Ethernet protocol is that the nodes need to be quite powerful and complex (and therefore more expensive) in order to handle the communication control. Another drawback with Ethernet is that during network traffic congestion the delay jitter can be severe and unpredictable, although at low network load Ethernet gives almost no delay.

Time-triggered communication on CAN

Traditional CAN communication is event based: asynchronous events are triggered by node applications that initialize each transmission session. In many cases, this strategy is an efficient way to share the network resource. There are a variety of applications, however, that require a guaranteed access to the communication bus with a fixed periodic rate. This constraint is typical for sampled-data feedback control systems. In the automotive industry, x-by-wire

systems are examples of such control systems with deterministic communication behavior during regular operation.

By introducing the notion of global network time, the standard ISO 11898-4 defines the extension *Time-triggered communication on CAN* (TTCAN). It is built on top of the traditional event-triggered CAN protocol and enables existing CAN nodes to work in parallel with TTCAN nodes. The global clock requires hardware implementation; otherwise, TTCAN is a pure software extension of CAN. The synchronization in TTCAN takes place through a periodic reference message, which all TTCAN nodes recognize and use to synchronize their clocks. The nodes are configured to know when to send their message after the reference message. The period time of the transmission of a periodic node should be a multiple of the reference period. Traditional CAN nodes (or event-based TTCAN nodes) compete for the access of the free windows between the reference messages, along the line of the conventional CAN protocol. This mechanism is thus the reason why time-triggered and event-triggered scheduling is possible simultaneously in TTCAN.

The sender of the reference message is obviously a crucial node in TTCAN to guarantee clock synchronization. Therefore, an automatic procedure is provided for letting another node take over if the reference sender fails, and taking the reference back when the original clock master recovers. It is possible to use an external clock, for example, from the global positioning system (GPS).

2.3 A brief history

The evolution of microelectronics paved the way for introducing distributed control systems in vehicles. In the early 1980s there was, however, no low-cost and standardized protocol that was suitable for real-time control systems. Therefore, as we stated before, in 1983 Kiencke started the development of a new serial bus system at Bosch, which was presented as CAN in 1986 at the SAE congress in Detroit [8]. The development of CAN was mainly motivated by the need for new functionalities, but it also substantially reduced the need for wiring. The Bosch CAN Specification 2.0 was published in 1991 and then two years later the CAN protocol was internationally standardized as ISO 11898-1. The need for higher-layer protocols was recognized early. In 1991, CANKingdom was introduced by Kvaser. DeviceNet, another higher-layer protocol, was introduced by Allen-Bradley in 1994, and CANopen by CAN in Automation (CiA) in 1995. CiA is an international users and manufacturers group, which was founded in 1992. Mercedes-Benz has been using CAN in its passenger cars since 1992. Originally, CAN was used only for engine control, but today there are a variety of CAN nodes not only for powertrain and chassis control but also for body electronics and infotainment systems. Many other car manufacturers base their control architecture on CAN, including BMW, Fiat, Renault, SAAB, Volkswagen, and Volvo. The CAN architecture for a Volvo passenger car is described in the next section. The notion of time-triggered protocols for real-time systems was introduced

by Kopetz and co-workers [10]. Time-triggered extensions of CAN were discussed in the late 1990s and early 2000s. This led to the standardization of TTCAN as ISO 11897-4 in 2004. Currently, there are intensive activities on utilizing TTCAN in a variety of vehicle applications.

3 Architectures

In this section, four vehicular examples of distributed control architectures based on CAN are presented. The architectures are implemented in a passenger car, a truck, a boat, and a spacecraft.

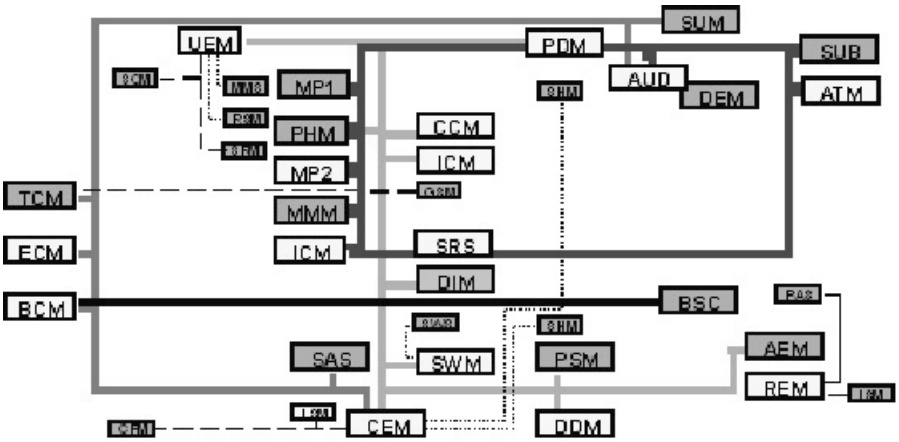
3.1 Volvo passenger car

In the automotive industry, there has been a remarkable evolution over the last few years in which embedded control systems have grown from standalone control systems to highly integrated and networked control systems. Originally motivated by reduced cabling and the specific addition of functionalities with sensor sharing and diagnostics, there are currently several new x-by-wire systems under development that involve distributed coordination of many subsystems.

Fig. 7 shows the distributed control architecture of the Volvo XC90. The blocks represent ECUs and the thick lines represent networks. The actual location of an ECU in the car is approximately indicated by its location in the block diagram. There are three classes of ECUs: powertrain and chassis, infotainment, and body electronics. Many of the ECU acronyms are defined in the figure. Several networks are used to connect the ECUs and the subsystems. There are two CAN buses. The leftmost network in the diagram is a CAN for power train and chassis subsystems. It connects for example engine and brake control (TCM, ECM, BCM, etc.) and has a communication rate of 500 kbps. The other CAN connects body electronics such as door and climate control (DDM, PDM, CCM, etc.) and has a communication rate of 125 kbps. The central electronic module (CEM) is an ECU that acts as a gateway between the two CAN buses. A media oriented system transport (MOST) network defines networking for infotainment and telematics subsystems. It consequently connects ECUs for multimedia, phone, and antenna. Finally, local interconnect networks (LINs) are used to connect slave nodes into a subsystem and are denoted by dashed lines in the block diagram. The maximum configuration for the vehicle contains about 40 ECUs [7].

3.2 Scania truck

There are several similarities between the control architecture in passenger cars and trucks. There are also many important differences, some of which

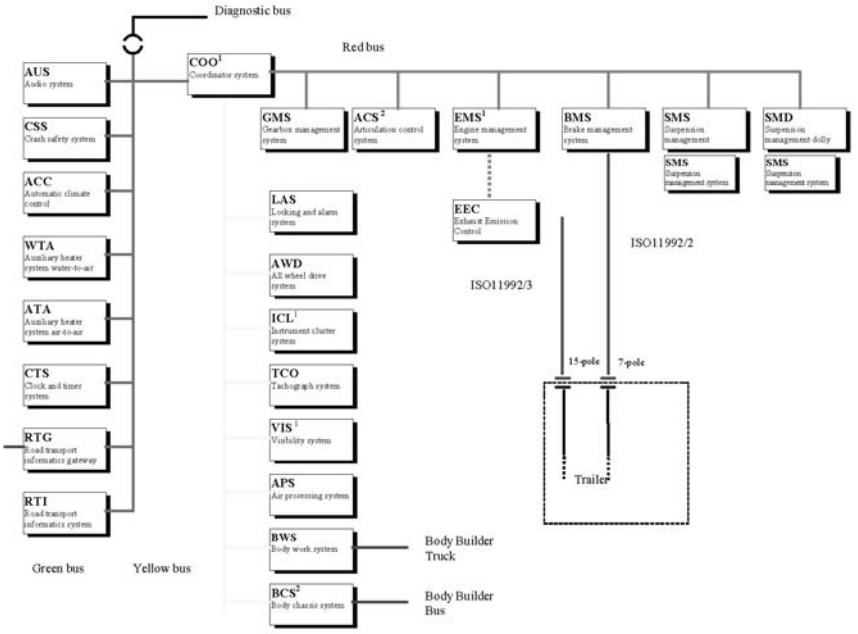


	Powertrain and chassis		Body electronics
TCM	Transmission control module	CEM	Central electronic module
ECM	Engine control module	SWM	Steering wheel module
BCM	Brake control module	DDM	Driver door module
BSC	Body sensor cluster	REM	Rear electronic module
SAS	Steering angle sensor	SWM	Steering wheel module
SUM	Suspension module	DDM	Driver door module
AUD	Audio module	PDM	Passenger door module
		REM	Rear electronic module
	Infotainment/Telematics	CCM	Climate control module
MP1,2	Media players 1 and 2	ICM	Infotainment control
PHM	Phone module	UEM	Upper electronic module
MMM	Multimedia module	DIM	Driver information module
SUB	Subwoofer	AEM	Auxiliary electronic
ATM	Antenna tuner module		

Fig. 7. Distributed control architecture for the Volvo XC90. Two CAN buses and some other networks connect up to about 40 ECUs. (Courtesy of Volvo Car Corporation.)

are due to the fact that trucks are configured in a large number of physical variants and have longer expected life times. These characteristics impose requirements on flexibility with respect to connecting, adding, and removing equipments and trailers.

The control architecture for a Scania truck is shown in Fig. 8. It consists of three CAN buses, denoted green, yellow, and red by Scania due to their relative importance. The leftmost (vertical) CAN contains less critical ECUs such as the audio system and the climate control. The middle (vertical) CAN handles the communication for important subsystems that are not directly involved in the engine and brake management. For example, connected to this



	Green bus		**Yellow bus**
AUS	Audio system	LAS	Locking and alarm system
CSS	Crash safety system	AWD	All wheel drive system
ACC	Automatic climate control	ICL	Instrument cluster system
WTA	Auxiliary heater water-to-air	TCO	Tachograph system
ATA	Auxiliary heater air-to-air	VIS	Visibility system
CTS	Clock and timer system	APS	Air processing system
RTG	Road transport info gateway	BWS	Body work system
RTI	Road transport info system	BCS	Body chassis system

	Red bus		
GMS	Gearbox management system	COO	Coordinator system
ACS	Articulation control system		
EMS	Engine management system		
EEC	Exhaust emission control		
BMS	Brake management system		
SMS	Suspension management system		
SMD	Suspension management dolly		

Fig. 8. Distributed control architecture for a Scania truck. Three CAN buses (denoted green, yellow, and red due to their relative criticality) connect up to more than twenty ECUs. The coordinator system ECU (COO) is a gateway between the three CAN buses. (Courtesy of Scania AB.)

bus is the instrument cluster system. Finally, the rightmost (horizontal) bus is the most critical CAN. It connects all ECUs for the driveline subsystems. The coordinator system ECU (COO) is a gateway between the three CAN buses. Connected to the leftmost CAN is a diagnostic bus, which is used to collect information on the status of the ECUs. The diagnostic bus can thus be used for error detection and debugging. Variants of the truck are equipped with different numbers of ECUs (the figure illustrates a configuration close to maximum). As for passenger cars, there are also subnetworks, but these are not shown in the figure.

SAE J1939 is the dominant higher-layer protocol for trucks. It facilitates plug-and-play functionality, but makes system changes and optimization difficult, partly because the priorities for scheduling the network traffic cannot be reconfigured. Manufacturers are using loopholes in SAE J1939 to work around these problems, but their existence indicates deficiencies in the protocol.

3.3 US Navy boat

There are several maritime applications of CAN. Here we give an example on unmanned seaborne targets provided by the United States Navy. The Navy has developed a distributed electronics architecture denoted SeaCAN, which is installed in all new seaborne targets and has been retrofitted into a number of older targets. A SeaCAN architecture for a 7 m remotely controlled rigid-hull inflatable boat is shown in Fig. 9. The system implements, for example, an autopilot based on a feedback control loop closed over the network. It involves the nodes Rudder Feedback, GPS Receiver, Pitch/Roll/Heading, Command/Control, and the two engine throttle nodes. The SeaCAN system uses a number of CPU boards with Infineon C167 microcontrollers connected together via a CAN bus running at 125 kbps. The lower communication rate is chosen to allow longer runs of copper and fiber suitable for larger boats and ships.

The SeaCAN system utilizes an operating system built around the CAN bus and based on the higher-layer protocol CANKingdom. The operating system contains a scheduler for tasks, which is synchronized with a higher-layer implementation of a global clock. It is thus possible to have coordinated behavior between two nodes, without any extra network communication. This functionality is used for generating periodic sampling, used in, e.g., the rudder servo control loop. In that case, the rudder sensing node samples the rudder angle at a rate of 10 Hz. The reception of the data messages from the rudder sensing node at the rudder actuator controller triggers the control loop routine. Because these messages have high priority on the bus and they are clocked out at a known rate, the control loop variability and data delay are very low, which thus enables a well-performing control loop.

Characteristics of SeaCAN include low usage of bandwidth in the order of 5%, global clock with a resolution of about 100 μsec, and special provisions with respect to safety and fail-safe shut-down. Network scheduling based on

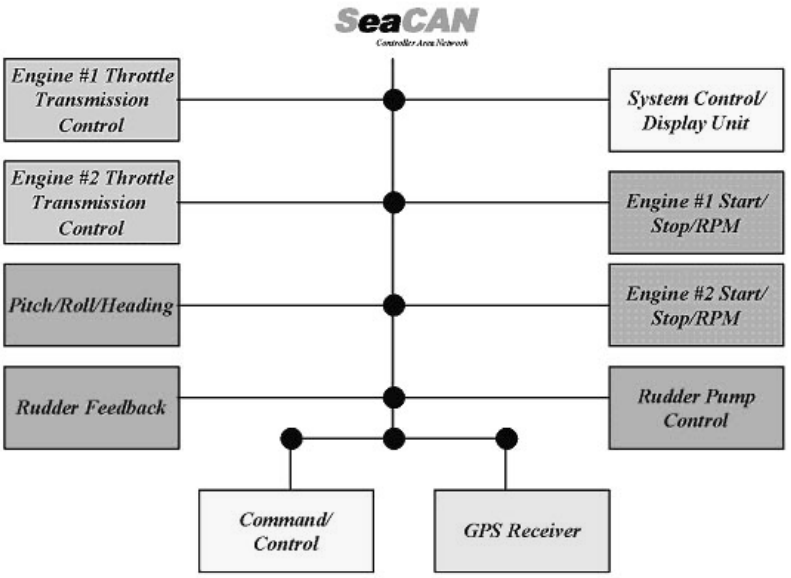


Fig. 9. Distributed control architecture for a boat. The block diagram shows a SeaCAN system for a 7 m remotely controlled rigid-hull inflatable boat. (Courtesy of the US Navy.)

the global clock is used to enforce a mixture of time-triggered and event-triggered communication.

3.4 SMART-1 spacecraft

The CAN protocol is also used in spacecraft and aircraft. SMART-1 is the first European lunar mission, where the acronym stands for "small missions for advanced research in technology." The spacecraft was successfully launched on September 27, 2003 by the European Space Agency on an Ariane V launcher. The Swedish Space Corporation was the prime contractor for SMART-1 and has developed several of the on-board subsystems including the on-board computer, avionics, and the attitude and orbit control system [2]. The main purpose of SMART-1 is to demonstrate the use of solar-electric propulsion in a low-thrust transfer from earth orbit into lunar orbit. The spacecraft carries several scientific instruments, and scientific observations are to be performed on the way to and in its lunar orbit. Currently (October 2004), SMART-1 is preparing for the maneuvers that will bring it into orbit [5].

Part of the distributed computer architecture of SMART-1 is presented in Fig. 10. The block diagram illustrates the decomposition of the system into two parts: one subsystem dedicated to the SMART-1 control system and

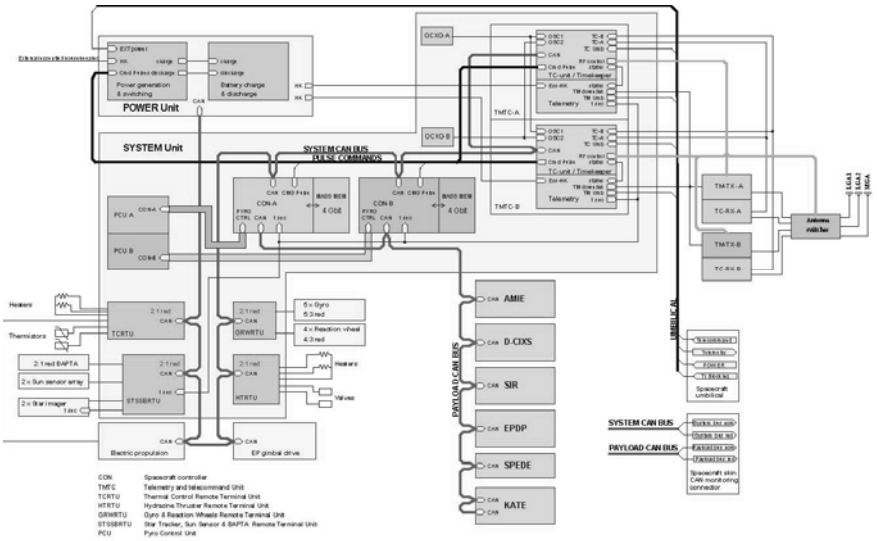


Fig. 10. Part of the distributed control architecture for the SMART-1 spacecraft. The system has two CAN buses: one for the control of the spacecraft and one for the payload. The spacecraft controllers are redundant and denoted CON-A and CON-B in the middle of the block diagram. (Courtesy of the Swedish Space Corporation.)

another for scientific experiments. Each of them is using a separate CAN, so there is one system CAN and one payload CAN. The spacecraft control is performed by the redundant controllers CON-A and CON-B, in the middle of the figure. Most control loops are closed over the system CAN with CAN nodes providing sensing and actuation capabilities. CAN nodes on the system bus include not only the spacecraft controller, but also nodes for telemetry and telecommand (earth communication), thermal control, star tracker and sun sensors, gyro and reaction wheels, hydrazine thruster, power control and distributions, and electronic propulsion and orientation.

Several provisions have been taken to ensure system robustness. All nodes are redundant; some in an active, others in a passive fashion. Each CAN bus has one nominal and one redundant communication path. A strategy and a hierarchy for error detection, redundancy management, and recovery have been defined. The spacecraft controller can take the decision to switch over to the redundant bus (but not back again). Most other nodes will check for the life sign message from the spacecraft controller. If the life sign is not available, the nodes will attempt to switch to the other bus. The power unit has highest authority in the autonomy hierarchy and will switch to the redundant spacecraft controller if the primary controller is considered to have failed. The power unit can also control the activation and deactivation of the other nodes. As a last means for recovery, ground can intervene manually.

Radiation tolerance and the detection of radiation-induced errors are crucial for SMART-1. It was not possible to use ordinary CAN controllers because they are not radiation tolerant. Instead, the license to use the VHDL-code for CAN was purchased from Bosch and was used to implement a CAN controller in a radiation-tolerant field programmable gate array (FPGA). Some other features were added to the CAN protocol simultaneously, such as specific error detection and error handling mechanisms. In addition, clock synchronization was added, so that the resolution of the global clock became better than 1 msec. Effects due to radiation can still cause corrupted frames. Therefore, one of the identifier bits was chosen to be used as an extra parity bit for the identifier field.

4 Control Applications

Two vehicular control systems with loops closed over CAN buses are discussed in this section. The first example is a vehicle dynamics control system for passenger cars that is manufactured by Bosch. The second example is an attitude and orbit control system for the SMART-1 spacecraft discussed in the previous section.

4.1 Vehicle dynamics control system

Vehicle dynamics control[4] systems are designed to assist the driver in over-steering, under-steering and roll-over situations [9, 15]. The principle of a vehicle dynamics control (VDC) system is illustrated in Fig. 11. The left figure shows a situation where over-steering takes place, illustrating the case where the friction limits are reached for the rear wheels causing the tire forces to saturate (saturation on the front wheels will instead cause an under-steer situation). Unless the driver is very skilled, the car will start to skid, meaning that the vehicle yaw rate and vehicle side slip angle will deviate from what the driver intended. This is the situation shown for the left vehicle. For the vehicle on the right, the on-board VDC will detect the emerging skidding situation and will compute a compensating torque, which for the situation illustrated is translated into applying a braking force to the outer front wheel. This braking force will provide a compensating torque and the braking will also reduce the lateral force for this wheel.

The VDC system compares the driver's estimated intended course, by measuring the steering wheel angle and other relevant sensor data, with the actual motion of the vehicle. When these deviate too much, the VDC will intervene by automatically applying the brakes of the individual wheels and also by controlling the engine torque, in order to make the vehicle follow the

[4] Also known as electronic stability program, dynamic stability control, or active yaw control.

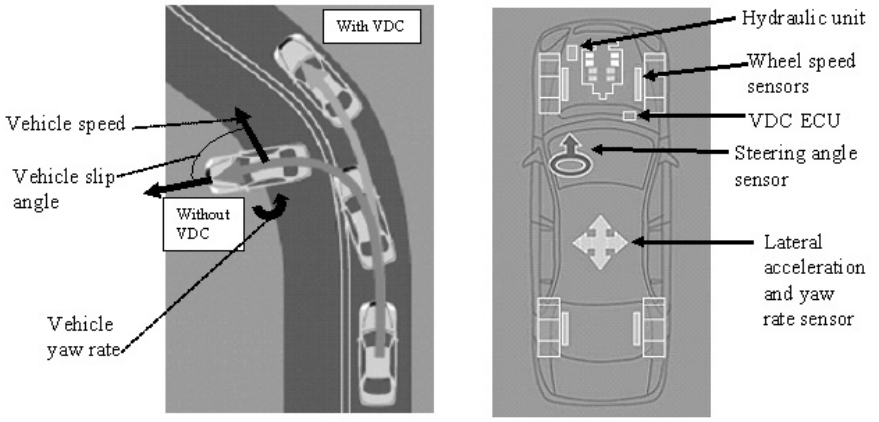


Fig. 11. Illustration of behavior during over-steering for vehicle with and without VDC system (left figure). Central components of VDC (right figure). (Based on figures provided by the Electronic Stability Control Coalition.)

path intended by the driver as closely as possible. The central components of VDC are illustrated on the right in Fig. 11. In essence, the VDC will assist the driver by making the car easier to steer and by improving its stability margin. See [9] for details.

A block diagram of a conceptual VDC is shown in Fig. 12. The cascade control structure consists of three controllers: (1) the yaw/slip controller, which controls the overall vehicle dynamics in terms of the vehicle yaw rate and the vehicle side slip angle; (2) the brake controller, which controls the individual wheel braking forces; and (3) the engine controller, which controls the engine torque. The inputs to the yaw/slip controller include the driver's commands: accelerator pedal position, steering wheel angle, and brake pressure. Based on these inputs and other sensor data, nominal values for the yaw rate and the vehicle side slip are computed. They are compared with the measured yaw rate and the estimated side slip. A gain-scheduled feedback control law is applied to derive set-points for the engine and brake controllers; for example, during over-steering, braking actions are normally performed on the front outer wheel and for under-steering normally on the rear inner wheel. The gains of the controllers depend on the driving conditions (e.g., vehicle speed, under-steering, over-steering). The brake and the engine controllers are typically proportional-integral-derivative (PID) controllers and also use local sensor information such as wheel speed. The VDC system has to take the driver behavior into account as well as disturbances acting on the vehicle, including cross-wind, asymmetric friction coefficients, and even a flat tire.

The VDC system utilizes the CAN bus, as it is depending on several ECUs, although the main functionality resides in a specific ECU. The implementation strongly depends on the choice of braking mechanics (e.g., hydraulics,

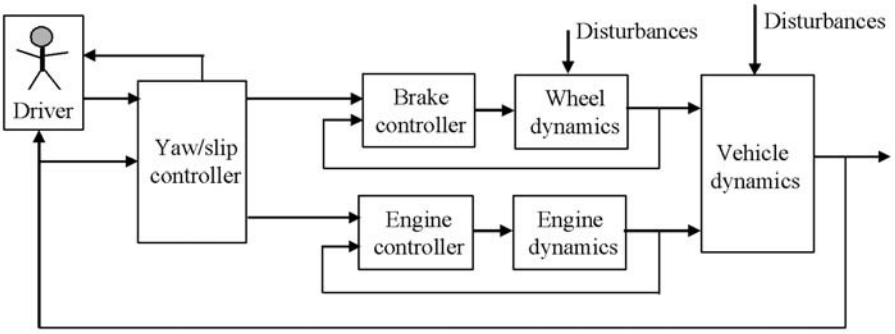


Fig. 12. Cascade control structure of VDC system

pneumatics, electro-hydraulics, or even electro-mechanics), the availability of a transmission ECU, and the interface to the engine ECU. A separate distributed control system is often used for the brakes, extending from a brake control node; for example, trucks often have one ECU per wheel pair and an additional controller for the trailer. Since some of the control loops of a VDC system are closed over a vehicle CAN, special care has to be taken with respect to end-to-end delays and faults in the distributed system.

4.2 Attitude and orbit control system

This section describe parts of the SMART-1 attitude and orbit control system and how it is implemented in the on-board distributed computer system [2]. The control architecture and the CAN buses of SMART-1 were described in Section 3. The control objectives of the attitude and orbit control system are to

- follow desired trajectories according to the goals of the mission,
- point the solar panels toward the sun, and
- minimize energy consumption.

The control objectives should be fulfilled despite the harsh environment and torque disturbances acting on the spacecraft, such as aero drag (initially when close to earth), gravitational gradient, magnetic torque, and solar pressure (mechanical pressure from photons). There are several phases that the control system should be able to handle, including the phase just after separation from the launcher, the thrusting phases on the orbit to the moon, and the moon observation phase.

The sensors and actuators used for controlling the spacecraft's attitude are illustrated in Fig. 13. Sensors are a star tracker and solid-state angular rate sensors. The star tracker provides estimates of the sun vector. It has one nominal and one redundant processing unit and two hot redundant camera

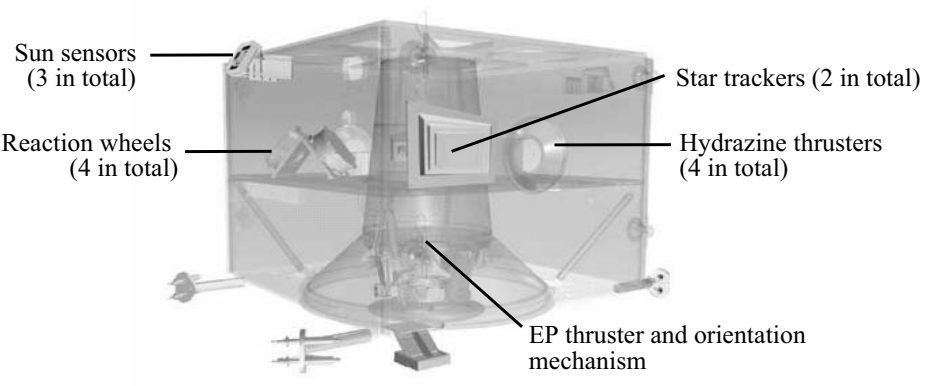


Fig. 13. Structure of SMART-1 spacecraft with sensors and actuators for the attitude and orbit control system. (Courtesy of the Swedish Space Corporation.)

heads, which can be operated from either of the two processing units. Five angular rate sensors are included to allow for detection and isolation of a failure in a sensor unit. The rate sensors can provide estimates of spacecraft attitude during shorter outages of attitude measurements from the star tracker. Actuators for the attitude control are reaction wheels and hydrazine thrusters. There are four reaction wheels aligned in a pyramid configuration based on considerations of environmental disturbances and momentum management. The angular momentum storage capability is 4 Nms per wheel with a reaction torque above 20 mNm. The hydrazine system consists of four nominal and four redundant 1 N thrusters.

The attitude and orbit control system consists of a set of control functions for rate damping, sun pointing, solar array rotation, momentum reduction, three-axis attitude control, and electric propulsion (EP) thruster orientation. The system has a number of operation modes, which consist of a subset of these control functions. The operation modes include the following:

- *Detumble mode:* In this mode, rotation is stabilized using one P-controller per axis with the aid of the hydrazine thrusters and the rate sensors.
- *Safe mode:* Here the EP thruster is pointed toward the sun and set to rotate one revolution per hour around the sun vector. The attitude is controlled using a bang-bang strategy for large sun angles and a PID controller for smaller angles. Both controllers use the reaction wheels as actuators and the sun tracker as sensor. The spacecraft rotation is controlled using a PI controller. When the angular velocity of the reaction wheels exceeds a certain limit, their momentum is reduced by use of the hydrazine thrusters.

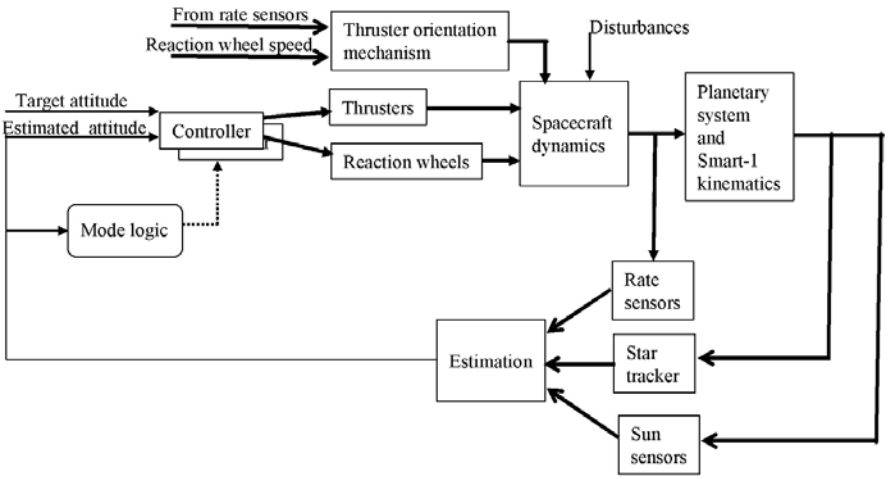


Fig. 14. Attitude control system under operation in Science mode. Disturbances include gravity, particles, and aero drag affecting the spacecraft.

- *Science mode:* In this mode, ground provides the attitude set-points for the spacecraft and the star tracker provides the actual attitude. The reaction wheels and the hydrazine thrusters are used.
- *Electric propulsion control mode:* This mode is similar to the science mode apart from the additional control of the EP orientation mechanism. This mechanism can be used to tilt the thrust vector in order to off-load the reaction wheel momentum about the two spacecraft axes that form the nominal EP thrust plane. This reduces the amount of hydrazine needed. The EP mechanism is controlled in an outer and slower control loop (PI) based on the speed of the reaction wheels and the rotation of the spacecraft body.

Let us describe the Science mode in some detail. Fig. 14 shows a block diagram of the attitude control system in Science mode. As was described for the Safe mode above, different controllers are activated in the Science mode depending on the size of the control error. This switching between controllers is indicated by the block "Mode logic" in the figure. Anti-windup compensation is used in the control law to prevent integrator windup when the reaction wheel commands saturate. Also, as in the Safe mode, the hydrazine thrusters are used to introduce an external momentum when the angular momentum of the reaction wheels grows too high. The attitude control works independently of the thruster commanding. The entire control system is sampled at 1 Hz. The time constant for closed-loop control is about 30 sec for the Science mode (and 300 sec for the Safe mode). The estimation block in Fig. 14 provides filtering of signals such as the sun vector and computes the spacecraft body rates. It includes a Kalman filter with inputs from the star tracker and

the rate sensors. The main purpose is to provide estimates of the attitude for short periods when the star tracker is not able to deliver the attitude, for example, due to blinding of the sensor camera heads. The control algorithms of the attitude and orbit control system reside in the spacecraft controllers of the control architecture depicted in Fig. 10. With a period of 1 sec, the spacecraft controller issues polling commands over the CAN to the corresponding sensors, including the gyros and the sun sensors. When all sensor data are received, the control commands are computed and then sent to the actuators, including the reaction wheels and the hydrazine thrusters. The maximum normal utilization of the CAN is about 30%, but under heavy disturbance, due to retransmission of corrupted messages, it rises to about 40%. The total communication time for communication over the CAN network for attitude and orbit control sensing and actuating data is approximately 12 msec. It is thus small compared to the sampling period.

The on-board software can be patched by uploading new software from ground during operation in space. So far this has been carried out once for the star tracker node. The need arose during a very intensive solar storm, which necessitated modification of software filters to handle larger disturbance and noise levels than had been anticipated.

During operation, the system platform software is responsible for detection of failing nodes and redundancy management. The control application is notified of detected errors such as temporary unavailability of data from one or more nodes. If the I/O nodes do not reply to poll messages for an extended period of time, the redundancy management will initiate recovery actions, including switching to the redundant slave nodes and attempting to use the redundant network.

5 Perspectives

The development of vehicles is going through a dramatic evolution, in their transition from pure mechanical systems to mechatronic machines with highly integrated hardware and software subsystems. DaimlerChrysler estimates that 90% of the innovations in the automotive area lie in electronics and software. A challenge in the development of vehicular embedded control systems is safety and real-time requirements. The control systems are increasingly being implemented in distributed computer systems and require a multitude of competences to be developed and integrated to meet quality requirements in a cost-efficient way. A major research problem is to develop techniques and tools to bridge the gap between functional requirements and the final design. In this section, we describe two particular trends in the vehicular networked embedded systems, namely, brake-by-wire and other x-by-wire systems, and standardized platforms and open-ended architectures for distributed control units in vehicles.

5.1 X-by-wire systems

X-by-wire is a term for the addition of electronic systems to the vehicle to improve tasks that were previously accomplished purely with mechanical and hydraulic systems. Examples of x-by-wire systems are steer-by-wire and brake-by-wire, where steering and braking data, respectively, are communicated electronically from the driver to the actuators. For a brake-by-wire control system, it is common that the information transfer from the brake pedal to the braking actuator is handled electronically, but that the actuators are hydraulic or pneumatic. Sometimes the actuators are replaced with electrical motors, so that a full brake-by-wire system is created.

An early x-by-wire system is the fly-by-wire application in the Airbus aircraft A310, which has been in use since 1983. Airbus 320, which was certified in 1988, is the first aircraft that depends entirely on x-by-wire control. Examples of early x-by-wire systems in the automotive industry include automated manual transmission, which eliminates the mechanical connection to the transmission, so that the gears can be chosen manually by pushing buttons or automatically by a computer that runs a gear selection program. The technology, which was first developed for motor sports to relieve the driver from using the clutch, is available from many passenger car manufacturers including Alfa Romeo, BMW, Mercedes-Benz, Porsche, and Volkswagen. Several solutions are also available for heavy vehicles, under names like I-Shift from Volvo Trucks and Opticruise from Scania. In the marine industry, a recent x-by-wire system is the throttle-by-wire system available from Mercury Marine, which is based on dual redundant CAN buses operating at 250 kbps.

Challenges for further applications of x-by-wire systems in the automotive industry are legislation, safety demands, cost efficiency, and user expectations. It should be noted that x-by-wire systems in cars are not the same as fly-by-wire systems. Differences include the sensitivity to frequency of failure in operations, and the cost sensitivity. Consequently, the technological concepts used for by-wire systems in airplanes are not necessarily cost efficient for the automotive industry.

5.2 Standardized software architectures

Standards in the automotive industry have been developed for communication, but the other parts of the distributed control systems of cars have mainly remained closed; for example, note that most automotive systems use proprietary real-time operating systems. The lack of existing standardization has generated research projects to standardize diagnostics and measurement systems, description languages, and software platforms. Two related initiatives focused on software platforms are OSEK/VDX [12] and AUTOSAR [1].

OSEK/VDX is a joint project of the automotive industry that aims at an industry standard for an open-ended architecture for distributed control units in vehicles. OSEK stands for *Offene Systeme und deren Schnittstellen für*

die Elektronik im Kraftfahrzeug (open systems and the corresponding interfaces for automotive electronics) and was founded by the German automotive industry. VDX, which is an acronym for vehicle distributed executive, was originally defined as part of a joint effort by the French companies PSA and Renault. In 1995 the OSEK/VDX group presented their first results of a specification. A goal of OSEK/VDX is to support the portability and reusability of application software. The open architecture defines three substandards of communication, network management, and operating system. It includes data exchange within and between ECUs, network configuration and monitoring, and real-time executive for ECU software. The basic idea of OSEK/VDX is to define standardized services, so that the costs are reduced to maintain and port application software. Obviously, by imposing portability, it should be possible to transfer application software from one ECU to another ECU. The application software module can have several interfaces. For example, there exist interfaces to the operating system for real-time control, but there also exist interfaces to other software modules. The interfaces should be rich enough to represent a complete functionality in a system. An OSEK/VDX implementation language has also been defined. It supports a portable description of all OSEK/VDX specific objects such as tasks and alarms. It remains to be seen whether the OSEK/VDX effort will be successful or not.

Automotive Open System Architecture (AUTOSAR) is a development partnership with the goal of standardizing basic system functions and functional interfaces in vehicles. The initiative is an indication of the difficulties faced today to fulfill the growing passenger and legal requirements, despite the increased system complexity. There needs to be a clear notion of how software components should be specified and integrated in automotive applications. The AUTOSAR standard is intended to serve as a platform upon which future vehicle applications could be implemented. The AUTOSAR project plan was released in 2003, so extensive test and verification remain to be done before AUTOSAR can be used in practice.

Acknowledgments

We would like to express our gratitude to the individuals and the companies that provided information on the examples described in this chapter. In particular, Per Bodin and Gunnar Andersson, Swedish Space Corporation, are acknowledged for providing information on the SMART-1 spacecraft. Dave Purdue, US Navy, is acknowledged for the description of SeaCAN. Jakob Axelsson, Volvo Car Corporation, is acknowledged for information on the Volvo XC90. Ola Larses, Scania AB, is acknowledged for information on the Scania truck. Lars-Berno Fredriksson is acknowledged for general advice on CAN.

References

1. http://www.autosar.org, 2004. Homepage of the development partnership Automotive Open System Architecture (AUTOSAR).
2. P. Bodin, S. Berge, M. Björk, A. Edfors, J Kugelberg, and P. Rathsman. The SMART-1 attitude and orbit control system: Flight results from the first mission phase. In *AIAA Guidance, Navigation, and Control Conference*, number AIAA-2004-5244, Providence, RI, 2004.
3. http://www.can-cia.de, 2004. Homepage of the organization CAN in Automation (CiA).
4. CAN specification version 2.0. Robert Bosch GmbH, Stuttgart, Germany, 1991.
5. http://www.esa.int/SPECIALS/SMART-1, 2004. Homepage of the SMART-1 spacegraft of the European Space Agency.
6. K. Etschberger. *Controller Area Network: Basics, Protocols, Chips and Applications*. IXXAT Automation GmbH, Weingarten, Germany, 2001.
7. J. Fröberg, K. Sandström, C. Norström, H. Hansson, J. Axelsson, and B. Villing. A comparative case study of distributed network architectures for different automotive applications. In *Handbook on Information Technology in Industrial Automation*. IEEE Press and CRC Press, 2004.
8. U. Kiencke, S. Dais, and M. Litschel. Automotive serial controller area network. In *SAE International Congress No. 860391*, Detroit, MI, 1986.
9. U. Kiencke and L. Nielsen. *Automotive Control Systems*. Springer-Verlag, Berlin, 2000.
10. H. Kopetz. *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, Dordrecht, 1997.
11. G. Leen and D. Heffernan. Expanding automotive electronic systems. *Computer*, 35(1):88–93, Jan 2002.
12. http://www.osek-vdx.org, 2004. Homepage of a joint project of the automotive industry on a standard for an open-ended architecture for distributed control units in vehicles.
13. M. Törngren. A perspective to the design of distributed real-time control applications based on CAN. In *2nd International CiA CAN conference*, London, U.K., 1995.
14. M. Törngren, K. H. Johansson, G. Andersson, P. Bodin, and D. Purdue. A survey of contemporary embedded distributed control systems in vehicles. Technical Report ISSN 1400-1179, ISRN KTH/MMK-04/xx-SE, Dept. of Machine Design, KTH, 2004.
15. A. T. van Zanten, R. Erhardt, K. Landesfeind, and G. Pfaff. VDC systems development and perspective. In *SAE World Congress*, 1998.

Control of Autonomous Mobile Robots

Magnus Egerstedt

School of Electrical Engineering, Georgia Institute of Technology, Atlanta, GA 30332, U.S.A., magnus@ece.gatech.edu

1 Background

An example of a reactive, mobile, embedded control system that has received considerable attention during the last decade is the *autonomous mobile robot*. The flurry of research activities in this area can be directly traced to the many exciting current and future applications where robotic systems are safer/cheaper/more effective than their human counterparts. Examples of current applications include

- Domestic service robots, such as autonomous vacuum cleaners, lawn mowers, and pool cleaners;
- Planetary exploration robots, such as the NASA Mars rovers Spirit and Opportunity;
- Autonomous robots for military applications, including surveillance and search-and-destroy robots; and
- Robots for monitoring, exploring, and securing unsafe environments, such as bomb sniffers, disaster site robots, and mine sweepers.

Notably absent from this list are the many robots employed in industrial settings. Such industrial robots are not considered here since they typically operate in highly structured environments, where the maneuvers can be planned in advance, resulting in challenging, yet standard, tracking problems. In contrast to this, autonomous mobile robots operate in partially or completely unknown environments, where the occurrences of unmodeled obstacles are commonplace. What this means is that the complexity of the control task is increased due to the complexity of the environment in which the system operates, which imposes a number of challenges on the control design. In this chapter we will cover modeling and architectural design issues for such systems. We will moreover discuss some implementation aspects as well as outline a collection of major challenges that still remain to be solved.

2 Multi-Modal Control

For mobile, autonomous robots the ability to function in and interact with a dynamic, changing environment is of key importance. As such, they fall under a class of reactive, mobile systems where environmental changes trigger changes in what objectives the control system must meet. The standard way of structuring the control system in order to deal with this problem is within a *multi-modal* control framework, sometimes referred to in the robotics literature as the *behavior-based* robotics framework [1, 4, 11]. The main idea is to identify different controllers, responses to sensory inputs, with desired robot behaviors. This way of structuring the control system into separate behaviors, dedicated to performing certain tasks, has gained significant momentum within the robotics community. This momentum stems from the fact that a modular design both simplifies the design process and also makes it possible to add new behaviors to the system without causing any major increase in complexity.

Once a collection of behaviors has been designed, different options present themselves at the supervisory level. For instance, one can let different behaviors run concurrently in the sense that they all can have an effect on the low-level motor commands according to some coordination rule. This construction with concurrent behaviors makes it relatively straightforward to stress robustness issues explicitly, since, for example, an "avoidance behavior" can just be given a higher priority or weight than a "reach target behavior." However, as multiple behaviors are allowed to affect the system simultaneously, a number of theoretical as well as practical issues present themselves. For example, given a collection of individual behaviors, how should these be coordinated in order to achieve a satisfactory, global behavior?

An alternate route to take is to insist on letting only one behavior affect the system at each time instant. This somewhat simplifies the analysis since the resulting system is a hybrid system for whose analysis a number of tools are available. But, as we will see, hard switches between behaviors may cause the performance of the system to degrade if care is not taken when dealing with chattering and other issues.

2.1 Behaviors

If we let the autonomous robot be modeled at the kinematic level as a unicycle

$$\begin{aligned}\dot{x} &= v\cos\phi \\ \dot{y} &= v\sin\phi \\ \dot{\phi} &= \omega,\end{aligned}$$

where (x, y) denotes the position of the robot, and ϕ denotes its orientation, a behavior is characterized by the way sensory data is mapped to the control inputs v and ω, corresponding to translational and angular velocities,

respectively. Now, relative to this robot model, a straightforward way [1] of specifying the effect of individual behaviors is to let the behavior define a vector

$$\mathcal{B} = r_{\mathcal{B}} \begin{pmatrix} \cos(\phi_{\mathcal{B}}) \\ \sin(\phi_{\mathcal{B}}) \end{pmatrix},$$

where $r_{\mathcal{B}}$ is the "magnitude" of the behavior vector, and $\phi_{\mathcal{B}}$ is its orientation. This vector formalism allows us to map behavior vectors to control values using some appropriate map $F(\mathcal{B}) = (v, \omega)^T$. For example, one can let

$$\begin{pmatrix} v \\ \omega \end{pmatrix} = F(\mathcal{B}) = \begin{pmatrix} \min\{v_0, 1/r_{\mathcal{B}}\} \\ C(\phi_{\mathcal{B}} - \phi) \end{pmatrix}.$$

Here, the translational velocity achieves its nominal value $v_0 > 0$ when the magnitude of the behavior vector is small, but is reduced as this magnitude grows. Furthermore, the angular velocity is simply given by a proportional error feedback law, with $C > 0$ being the gain. Note that it is also quite standard to let the gain vary as a function of $r_{\mathcal{B}}$.

Now, if we are given $\mathcal{B}_1$ and $\mathcal{B}_2$, i.e., two vectors corresponding to two different behaviors, they can be combined directly using a vector addition operation $\mathcal{B}_1 + \mathcal{B}_2$ in order to produce a new behavior, and this semigroup property is why the vector notation is particularly appealing. Here the coordination mechanism is thus explicitly given. Moreover, the magnitude of the behavior vector, $r_{\mathcal{B}}$, is what determines how much weight that particular behavior is given in the summation. As we will see in the next few paragraphs, avoidance behaviors should increase in magnitude, typically according to an inverse square law, as the robot draws closer to the obstacles.

To make matters more concrete, let us in consider an obstacle-avoidance behavior (denoted OA in what follows) in some detail. Most mobile robots are equipped with a collection of k range sensors, such as ultrasonic sonars or infrared sensors, and a standard sonar ring typically consists of 8 or 16 sensors. Each of these sensors measures the distance to the closest obstacle along a particular, fixed relative orientation; we let d_j denote the distance to the closest obstacle detected by sensor j, and we let ϕ_j be the corresponding angle. We can then define the obstacle avoidance behavior, $\mathcal{B}_{OA}$, through the vector summation

$$\mathcal{B}_{OA} = \mathcal{B}_{OA,1} + \mathcal{B}_{OA,2} + \ldots + \mathcal{B}_{OA,k},$$

where the behavior vectors are given by

$$r_{\mathcal{B}_{OA,j}} = \begin{cases} 0 & \text{if } d_j > D \\ C_{OA} \frac{(D - d_j)}{d_j^3} & \text{otherwise} \end{cases}$$
$$\phi_{\mathcal{B}_{OA,j}} = \pi + \phi_j,$$

where $C_{OA} > 0$, and D is the safety distance at which the obstacle-avoidance behavior starts affecting the system.

In a similar manner, we can define an "approach target behavior," $\mathcal{B}_{AT}$, as

$$r_{\mathcal{B}_{AT}} = C_{AT}$$
$$\phi_{\mathcal{B}_{AT}} = \arctan((y_g - y)/(x_g - x)),$$

where $C_{AT} > 0$ is the constant magnitude, and the goal is located at (x_g, y_g), as shown in Fig. 1.

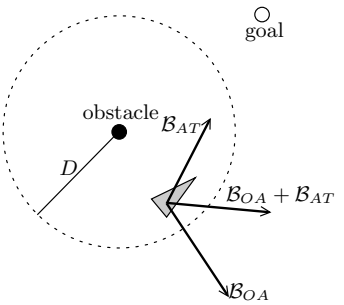


Fig. 1. Vector addition of "obstacle-avoidance" and "approach target" behaviors

2.2 Regularizations

However, it may not always be desirable to let different behaviors affect the system simultaneously, even though such an approach results both in notational convenience as well as an intuitively appealing mechanism for combining multiple control objectives. Unfortunately, such an approach ruins the modularity that comes with a switched control strategy in the sense that if a new behavior is introduced, its impact on the system is almost impossible to characterize analytically. This lack of analytical characterization tools is one of the main reasons why *emergent behaviors*, i.e., unpredictable global behaviors obtained through local rules, have received considerable attention in the literature. Moreover, if an obstacle-avoidance behavior has been designed so that the robot is guaranteed not to hit static obstacles, by combining this behavior with other behaviors, this guarantee no longer holds.

A remedy to this problem is to let the control system switch between different behaviors. Unfortunately, such an approach may have a negative impact on the performance of the system since it increases the risk of introducing chattering into the system. Chattering is a phenomenon that occurs when two vector fields, corresponding to two different behaviors, both point in toward the switching surface that dictates when the robot should switch between the behaviors. In other words, if we switch from mode 1, where $\dot{x} = f_1(x)$, to mode 2, where $\dot{x} = f_2(x)$, when x leaves the region $g(x) < 0$, where g is a smooth map from $\mathbb{R}^n$ to $\mathbb{R}$, then chattering occurs if

$$\frac{\partial g(x)}{\partial x} f_1(x) > 0 \quad \text{and} \quad \frac{\partial g(x)}{\partial x} f_2(x) < 0$$

on the boundary $g(x) = 0$.

The standard way out of this problem is to regularize the system so that sliding is allowed to occur. For example, assume that we have access to instantaneous heading control in our control laws. When an obstacle is closer to the robot than D, the obstacle-avoidance behavior is active. Since the repulsive potential field from that behavior will be orthogonal to the surface on which the behavior becomes active, the sliding solution is given by

$$\mathcal{B}_S = \alpha \mathcal{B}_{OA} + (1 - \alpha)\mathcal{B}_{AT},$$

for some $\alpha \in [0, 1]$ such that $\mathcal{B}_S \perp \mathcal{B}_{OA}$.

Some results from applying this regularization approach to the chattering problem are shown in Fig. 2, where (a) shows a situation when vector summation is used. Fig. 2(b) corresponds to switches between the behaviors, and it is clear that a chattering-like behavior is produced. In (c), the regularized solution is shown. Even though we only have one behavior active at a time, the performance is clearly satisfactory in that case.

(a) (b) (c)

Fig. 2. (a) Combined behaviors using vector summation, (b) switches between the different behaviors, and (c) a regularized solution. These results were obtained on the Nomad simulator, the Nserver.

By incorporating this type of information about the different behaviors, it is possible to generate the sliding modes automatically. It furthermore suggests that this method would scale when more than two behaviors affect the motion of the robot, as long as an automatic procedure for designing the sliding solutions can be identified. Hence we assume throughout the remainder of this chapter that only one behavior affects the robot at each time instant, and that, when appropriate, a sliding mode may be induced from the system dynamics. In the next section we will consider this issue to be settled and instead focus on the question of how one should model and specify multimodal control procedures for robotics tasks.

3 Task Specifications

As an example, consider the scenario that is played out in Fig. 3. The problem the robot is instructed to solve is to find the door while avoiding obstacles. Once the door is found, the robot should go though the door and then start following the wall.

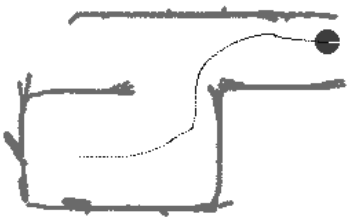


Fig. 3. The robot finds the door and then moves into a corridor, where it follows the wall. The reason why the sonar readings seem rather inaccurate is due both to the odometric drift and the coarseness of the sonar resolution.

3.1 Hybrid automata

There are, of course, a number of ways in which this particular navigation problem can be solved. If we let our solution lay within the multi-modal framework, we first note that the behaviors needed to solve this problem could be

- Wander;
- Avoid obstacle;
- Go through door; and
- Follow wall.

Moreover, it is clear that by designing the individual behaviors, a complete, executable multi-modal control procedure has not yet been produced in the sense that we cannot run it directly on the robot. What is missing is the set of rules for switching between the different behaviors. For instance, if we let the dynamics of the system be

$$\begin{aligned} \dot{x} &= f(x,u),\ x \in \mathbb{R}^n,\ u \in U \subset \mathbb{R}^m \\ y &= h(x),\ y \in Y \subset \mathbb{R}^p, \end{aligned}$$

then the different behaviors correspond to particular feedback laws $u_{\mathcal{B}} : Y \to U$. If we let u_1 and u_2 be two such behaviors, then, in mode 1, we have $\dot{x} = f(x, u_1(h(x)))$ and, in mode 2, $\dot{x} = f(x, u_2(h(x)))$. We moreover choose to

switch between these two modes as a certain property, defined on the sensory inputs, is satisfied. We can thus define a Boolean map $\xi : Y \to \{0, 1\}$ and use mode 1 as long as $\xi(y) = 0$, and switch to mode 2 when $\xi(y) = 1$. Such a mapping is referred to as a *guard* in the hybrid systems literature, and in Fig. 4, a *hybrid automaton* is shown that generates the desired switched behavior. The interpretation here is that the system starts at the initial condition $x = x_0$ in mode 1, and switches to mode 2 as the guard condition is satisfied.

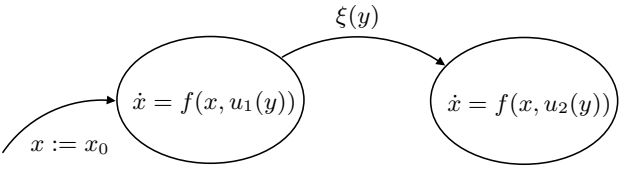


Fig. 4. A simple hybrid automaton

If we now return to the problem of having the robot find the door and then go through it, a hybrid automaton that would implement a solution to this problem is shown in Fig. 5, where u_W corresponds to the "wander" behavior, u_O corresponds to "obstacle avoidance," u_D corresponds to "go through door," and u_F to "follow wall." Moreover, $\xi_O = 1$ if an obstacle is too close, while $\xi_S = 1$ when this is no longer the case. $\xi_D = 1$ when the door has been detected, and $\xi_T = 1$ when the robot has passed through the door.

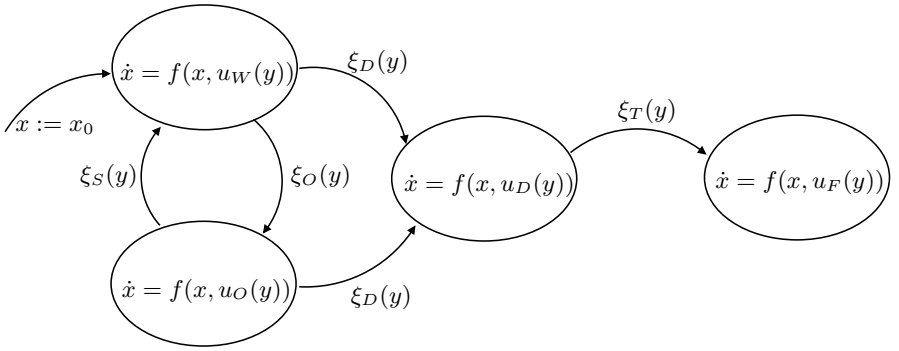


Fig. 5. A hybrid automaton for the scenario in Fig. 3

The hybrid automaton in Fig. 5 captures the switched aspects of a multi-modal control procedure in a very natural way. As such, it has been the main model used when analyzing the performance of autonomous mobile robots governed by reactive, multi-modal control strategies. Formally, a hybrid automaton can be defined as $H = (Q, X, f, E, \Xi, x_0, q_0)$, where $Q = \{q_1, q_2, \ldots, q_N\}$ is the set of discrete states (or modes). X is the state space on which the

continuous state is evolving, e.g., $X = \mathbb{R}^n$, and $f : X \times Q \to TX$ is the mode-dependent differential equation that dictates the evolution of the continuous state. Moreover, $E \subset Q \times Q$ is the edge set, and Ξ is the set of guards, with $\Xi \ni \xi : X \times E \to \{0, 1\}$. (We will suppress ξ's dependence on $e \in E$ whenever this dependence is clear from the context.) Finally, q_0 and x_0 denote, respectively, the initial discrete mode and continuous state [8].

Due to its expressiveness, the hybrid automaton is a useful modeling and analysis tool. However, from a specification and implementation point of view, we note that it may be more beneficial to use a more compact notation, which leads us to the idea of a formal language. (See, for example, [9] for an introduction to this subject.)

3.2 Motion description languages

We are given a finite automaton $A = (Q, \Gamma, \delta, q_0)$, where Q (finite) is the state space, Γ (finite) is the event set, q_0 is the initial condition, and $\delta : Q \times \Gamma \to Q$ is the transition function. We say that A generates the *language* $L(A)$, where

$$L(A) = \{s \in \Gamma^\star \mid \delta(q_0, s) \text{ is defined } \}.$$

Here $\Gamma^\star$ is the free monoid over Γ, i.e., the set of all finite length words over Γ (including the empty word ϵ). In other words, $\Gamma^\star$ is the set of all finite length concatenations of events, and if $s = \gamma_1 \cdot \gamma_2 \cdots \gamma_N$ then $\delta(q_0, s) = \delta(\cdots \delta(\delta(q_0, \gamma_1), \gamma_2), \cdots, \gamma_N)$.

Consider, for example, the case where we ignore the differential equations in the automaton in Fig. 5, and use ξ and u as shorthand for the event $\xi(y) = 1$ and the mode corresponding to the behavior $\dot{x} = f(x, u)$, respectively. Then $Q = \{u_W, u_O, u_D, u_F\}$, $\Gamma = \{\xi_O, \xi_S, \xi_D, \xi_T\}$, $q_0 = u_W$, and $\delta(u, \xi)$ is given by the discrete transitions in the automaton. In this case, the generated language becomes

$$L(A) = \overline{\overline{\{(\xi_O \cdot \xi_S)^\star\}} \cdot \{\xi_D \cdot \xi_T\}},$$

where, given a language $L \subset \Gamma^\star$, $\overline{L} = \{s \in \Gamma^\star \mid l = s \cdot t \in L \text{ for some } t \in \Gamma^\star\}$ is the *prefix closure* of the language L, the language *concatenation* $L_1 \cdot L_2 = \{s \in \Gamma^\star \mid s = l_1 \cdot l_2 \text{ for some } l_1 \in L_1,\ l_2 \in L_2\}$, and, given $s \in \Gamma^\star$, $s^\star = \{\epsilon, s, s \cdot s, s \cdot s \cdot s, \ldots\}$, where ϵ is the empty word.

Now, given A and a string $s \in L(A)$, this string completely determines the evolution of A, as long as A is deterministic. However, it does not capture the structure of A itself. In other words, from $L(A)$ we can analyze the evolution of A, but as a tool for specifying and implementing finite automata it falls short. And, since we in this section are interested in finding compact yet expressive ways for implementing and specifying multi-modal control procedures, where each discrete state corresponds to a particular mode, the specification language must include a description of what behaviors to use as well as what guard relations should characterize the mode transitions.

One such language that has appeared in the literature is the *motion description language*. Given a finite set of behaviors B (or more precisely, symbols that correspond to behaviors) and interrupts Ξ, we can let a motion description language be a subset of the set $(B \times \Xi)^\star$, i.e., a multi-modal control procedure contains strings of behavior-guard pairs, where the behavior specifies what control law to use, while the guard (or interrupt) dictates under which conditions this behavior should terminate. Using these ideas (and using the convention that u is a symbol corresponding to the map $u : Y \to U$), the multi-modal control procedure implemented in Fig. 5 is

$$(((u_W, \xi_O) \cdot (u_O, \xi_S))^\star, \xi_D) \cdot (u_D, \xi_T) \cdot (u_F, \xi_\epsilon),$$

where $\xi_\epsilon(y) = 0,\ \forall y \in Y$. The interpretation here is that the "meta-behavior" $(u_W, \xi_O) \cdot (u_O, \xi_S)$ is repeated until $\xi_D(y) = 1$, followed by the string $(u_D, \xi_T) \cdot (u_F, \xi_\epsilon)$.

3.3 Complexity issues

Given that motion description languages are used for specifying and implementing multi-modal control procedures on embedded robotics systems, one has access to a formalism in which control procedures can be thought of as having an information theoretic content. In other words, they need to be coded using a certain number of bits. This is facilitated by the fact that a behavior-interrupt pair corresponds to a particular symbol in a finite set. Each such symbol thus represents a different control actions that, when applied to a specific machine, define a particular segment of motion [3, 6, 10, 12].

Now, assume that we are given a string of modes $s \in (B \times \Xi)^\star$. The number of bits needed for describing s uniquely is given by the *description length*:

$$\mathcal{D}(s, B \times \Xi) = |s| \log_2(card(B \times \Xi)),$$

where $|s|$ denotes the length of s, i.e., the number of modes in the string, and $card(\cdot)$ denotes cardinality. The description length thus tells us how complicated s is, i.e., how many bits we need for describing it.

Now, since Y corresponds to the set in which the sensor readings take on their values, and since every physical sensor has a finite range and resolution, we can assume that Y has finite cardinality. A similar argument can be applied on the actuator side as well, and hence we can assume that both Y and U are finite sets, which is the case in the emerging area of *quantized control* as well [5].

Moreover, since

$$card(B \times \Xi) = card(U)^{card(Y)} 2^{card(Y)},$$

we see directly that a higher resolution measurement results in a larger Y (and hence in a potentially higher description length) than a lower resolution

measurement does. A better sensor might thus make the control procedures significantly more complicated while only providing marginally better performance. This trade-off between complexity and performance is something that can be capitalized on when designing control laws. The idea is simply (in theory) to pick $(U, Y, s \in (B \times \Xi)^\star)$ in such a way as to make the robot behave satisfactorily, while minimizing the description lengths of the control procedures.

For example, in [6], a quantitative analysis was given, showing that the availability of feedback can reduce the length of the shortest description of the multi-modal control procedures. In particular, it was shown that the length of the description can be reduced by a factor that depends on the ratio of the size of the entire state space to the size of the set of states for which feedback is locally effective, i.e., where convergent observers can be constructed. In other words, in some situations it is better to rely on sensors than to work solely with map-based open-loop instructions if one wants to minimize the description length of the multi-modal control procedure.

This result holds the promise of further generalizations since it can be thought of as a special case of the *sensor selection problem* (what sensors are needed to carry out the navigation task successfully?), as it tells us whether or not sensors should be used at all. Furthermore, the many visible and successful applications of feedback mechanisms at work testify to their effectiveness and, over the years, various arguments have been advanced showing why, in particular settings, feedback is useful. The models commonly used bring to the fore considerations of sensitivity, uncertainty, etc., and specific formalizations include

1. H.S. Black's argument for reducing the effect of drift in a high-gain amplifier by the use of a relatively constant, but low gain, feedback term;
2. The stochastic disturbance argument for using measurements to reduce the effect of probabilistic uncertainty; and
3. The game theoretic argument in which a saddle point condition is enforced by feedback. (H_∞ control can be thought of this way.)

To this list a fourth item has thus been added that can be cast in terms of the effect feedback has on reducing the description length of the control procedures for robot navigation tasks:

4. The complexity argument, showing that feedback can shorten the description of the control procedure if reliable sensory information is available.

Some extensions to this work were undertaken in [7] based on the observation that goals are seldom final goals. More often they tend to be intermediary goals in a grander scheme, which, for instance, is the case when mobile robots are navigating using sequences of landmarks. A collection of results was derived that describe how the complexity of the input signals decrease if the robot is navigating in an environment populated by many, easily detectable landmarks.

However, the description length does not tell the whole story. If we assume that we have been able to establish a probability distribution over $B \times \Xi$, we can use optimal coding schemes, such as the Huffman code, for finding the shortest expected number of bits $l^\star(B \times \Xi)$ needed for coding an element drawn at random from $B \times \Xi$. Shannon's classic source coding theorem tells us that $\mathcal{H}(B \times \Xi) \leq l^\star(B \times \Xi) < \mathcal{H}(B \times \Xi) + 1$, where the *entropy* $\mathcal{H}(B \times \Xi)$ is given by

$$\mathcal{H}(B \times \Xi) = - \sum_{i=1}^{card(B \times \Xi)} p_i \log_2 p_i.$$

Here the interpretation is that the behavior-interrupt pair $s_i \in B \times \Xi$ occurs with probability p_i, and it should be noted that a probability distribution over $B \times \Xi$ corresponds to a specification of which modes are potentially useful. But, to establish such a probability distribution over a structured set, such as the set of modes, is not a trivial task, and only initial work has been conducted along these lines [2].

4 Final Remarks

The main focus of this chapter has been on the modeling and specification issues that arise when dealing with autonomous mobile robots. In particular, a multi-modal control framework has been promoted as a way to decompose the control system into a collection of building blocks. The resulting hybrid control system is particularly suited to the reactive character of the navigation system that complex and unknown environments inevitably give rise to. However, there are a number of challenging research issues that remain largely unsolved and need to be addressed. To conclude this chapter, we present a partial list over such areas of particular importance in robotics research.

- *From local rules to global behaviors:* Given a multi-modal control procedure (or a class of such procedures), what can be said about the resulting system in terms of stability, robustness, expressiveness, and task completion?
- *From global behaviors to local rules:* Given a desired overall behavior, what local behaviors, or modes, are needed in order to achieve the global task?
- *Adaptive multi-modal control:* Given a collection of behaviors and guards, can they be adaptively varied over time in order to achieve a better overall performance?
- *Computational complexity versus real time:* In robotics, a distinction is made between *deliberative* and *reactive* behaviors, where the former relies on internal representations of the environment, which facilitates the use of planning of optimal paths, etc. Given constraints on the reaction time, can a trade-off be achieved between the complexity of the control algorithm and the time in which the algorithm has to terminate with an answer?

References

1. R. C. Arkin. *Behavior Based Robotics*. The MIT Press, Cambridge, MA, 1998.
2. A. Austin and M. Egerstedt. Mode reconstruction for source coding and multimodal control. *Hybrid Systems: Computation and Control*, Springer-Verlag, Prague, The Czech Republic, April 2003.
3. R. W. Brockett. On the computer control of movement. In the *Proceedings of the 1988 IEEE Conference on Robotics and Automation*, pp. 534–540, New York, April 1988.
4. R. Brooks. A robust layered control system for mobile robots. *IEEE Journal of Robotics and Automation*, Vol. RA-2, No. 1, pp. 14–23, 1986.
5. D. F. Delchamps. The stabilization of linear systems with quantized feedback. *Proceedings of the IEEE Conference on Decision and Control*, Austin, TX, Dec. 1988.
6. M. Egerstedt and R. W. Brockett. Feedback can reduce the specification complexity of motor programs. *IEEE Transactions on Automatic Control*, Vol. 48, No. 2, pp. 213–223, Feb. 2003.
7. M. Egerstedt. Some complexity aspects of the control of mobile robots. *American Control Conference*, Anchorage, AK, May 2002.
8. T. A. Henzinger. The theory of hybrid automata. *Proceedings of the 11th Annual Symposium on Logic in Computer Science (LICS)*, pp. 278–292, IEEE Computer Society Press, Washington D.C., 1996.
9. J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation*, 2nd edition. Addison Wesley, Reading, MA, 2000.
10. D. Hristu-Varsakelis and S. Andersson. Directed graphs and motion description languages for robot navigation and control. *Proceedings of the IEEE Conference on Robotics and Automation*, May 2002.
11. D. Kortenkamp, R. P. Bonasso, and R. Murphy, Eds. *Artificial Intelligence and Mobile Robots*. The MIT Press, Cambridge, MA, 1998.
12. V. Manikonda, P. S. Krishnaprasad, and J. Hendler. Languages, behaviors, hybrid architectures and motion control. In *Mathematical Control Theory*, Eds. Willems and Baillieul, pp. 199–226, Springer-Verlag, Berlin, 1998.

Wireless Control with Bluetooth

Vladimeros Vladimerou[1] and Geir Dullerud[2]

[1] Electrical and Computer Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801, U.S.A.
vladimer@uiuc.edu
[2] Mechanical and Industrial Engineering
University of Illinois at Urbana-Champaign
Urbana, IL 61801, U.S.A.
dullerud@uiuc.edu

Summary. This chapter introduces and summarizes the specifications of the Bluetooth wireless protocol and is intended for control engineers wanting to implement Bluetooth networking. It explains the basics of forming connections and networks in the context of a control application. The chapter also includes specific examples and a list of useful resources and methods of acquiring more information about Bluetooth.

1 What is Bluetooth?

A cable-replacement technology

Simply put, Bluetooth is a wire-replacement technology. It is an open-stack protocol, designed by Ericsson and adopted by thousands of companies [1], including Intel, Microsoft, IBM, Motorola, and Nokia. It is meant for short-range wireless applications with a capacity of 1Mbps. Cables for peripherals such as printers, computer mice, personal digital assistants (PDAs), or even headphones and refrigerators can all be replaced by a Bluetooth channel.

Usually, there is a host device, like a computer or PDA, which connects via an external Bluetooth board to the peripheral, which also has a Bluetooth board, most likely internal, as shown in Fig. 1. The Bluetooth boards[1] themselves have a Bluetooth **chipset**, which has the firmware and radio frequency (RF) hardware along with an antenna.

[1] A Bluetooth "board" is usually referred to as a "controller" and sometimes as a "dongle" or a "module." As defined in the Bluetooth specification, it is a subsystem containing the Bluetooth RF, baseband, resource controller, link manager, device manager, and a Bluetooth HCI (Hardware Controller Interface).

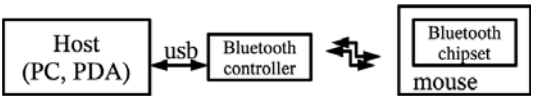


Fig. 1. An example Bluetooth link

Networks formed by Bluetooth devices are called Personal Access Networks (PANs). The "role" of a device in a PAN is either "a master" or "a slave." A PAN **must** have a master device. The master device controls access to all channels in the network and it forms what is called a "piconet". Slave-to-slave communication is not supported. Such communication can only be established if the common master relays packets from slave to slave. As shown in Fig. 2, when a slave device is also a master on its own piconet, a "scatternet" is formed – the combination of the two piconets. Communication is done between a master and a slave but not between the slaves. More explicitly, an example of a scatternet would be a Bluetooth-enabled PC (master) connected with a Bluetooth mouse (slave) and keyboard (slave) but also with a PDA (slave, master) which in turn connects to a headset (slave) on its own piconet. There can be up to 7 active slaves in a piconet. Bluetooth actually allows for slave devices to be in a "parked" state, from which they can recover in less time than the time required to establish a connection to the piconet from scratch.

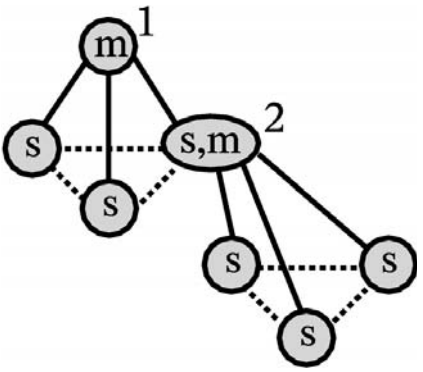


Fig. 2. Two piconets forming a scatternet. Controller 2 is the master in the bottom piconet and a slave in the top piconet.

Specifications

Bluetooth can generally connect devices that are in the same room. There are three classes for transmitting power in Bluetooth: 1 mW,[2] 2.5 mW, and 100 mW. The 1 mW-enabled devices are strong enough to transmit to around

[2]milliWatt

10 m or more, in an unobstructed environment. The third class reaches 100 m. The power class can be selectable on some Bluetooth chipsets.

The full specification is available on the web (see [2]), and so we only list some basic facts and aspects important for control applications. One can find several articles (for instance, [3]) and a plethora of web-pages on Bluetooth, its specifications, and its applications. A very good place to start would be the Palo Wireless website [4].

The Bluetooth access band is at 2.4 GHz, which sometimes causes interference with 802.11b networks. Although the authors have never seen, in practice, noticeable degradation in the performance of either 802.11b or Bluetooth when they are working simultaneously, there have been reports with contrary results. In any case, the protocol most likely to be affected would be 802.11b and not Bluetooth. This is due to the fact that 802.11b uses spread spectrum encoding whereas Bluetooth utilizes frequency hopping. The latter, when interfered with, is likely to merely degrade in bandwidth but unlike CDMA in 802.11b it will still be "on" (802.11b networks might stop functioning completely). Bluetooth uses forward error correction (FEC) and Gaussian frequency shift keying (GFSK) for frequency hopping (1600 hops/second).

The master device always specifies the hopping sequence as well as the access times for each of its slaves. That is why a slave device might not function as expected when it has the "slave" role in multiple piconets.

Layers

Fig. 3 shows the lower layers of the Bluetooth specification. The Hardware Controller Interface (HCI) layer provides access to all the hardware. It transfers commands and data from the host microprocessor to the controller chipset via universal serial bus (USB), universal asynchronous receiver and transmitter (UART), PC-Card, and other interfaces.

Based on commands and settings from the host, the HCI firmware interacts with the link manager (LM) firmware and baseband controller to scan for other devices, establish connections, and transmit and receive data. Fig. 4 shows how the HCI transfers data to and from the hardware.

In addition, most Bluetooth chipsets on the market (CSR Bluecore, Ericsson ROK) have their own customized HCI commands that are applicable only on a specific brand. These commands are always used in addition to the standard HCI commands conforming with the standard.

Pulse code modulated (PCM) data can also be sent directly to the lower layers for transmission. All layers above the HCI device driver are software. The HCI layer is implemented in both firmware and software, on the Bluetooth chipset and the host computer, respectively. The Logical Link Control and Adaptation Protocol (L2CAP) layer handles connections of Synchronous Connection-Oriented (SCO) and Asynchronous Connection-Oriented Link (ACL) types.

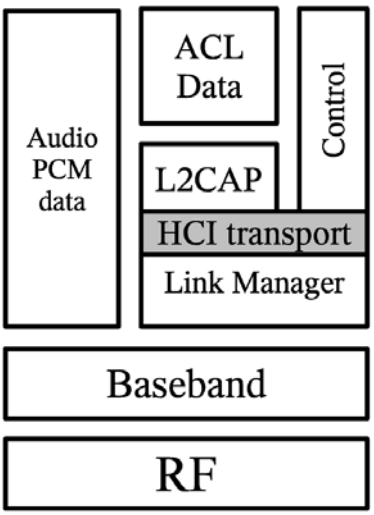


Fig. 3. The lower layers of the Bluetooth specification

Functions

Above the L2CAP layer several application-specific layers exist. The specification calls them "profiles." Such is the RFCOMM, a serial cable emulation protocol and HID which handles connections to human interface devices (mice, keyboards). Even the Internet protocol (IP) stack can be implemented on top of L2CAP. The 1.2 version of the specification defines many more profiles such as General Access Video Distribution and others.

The reader is encouraged to read through the different volumes of the specification [2] on Bluetooth profiles. At the time the authors first considered Bluetooth for wireless communication in controls, the HCI layer was the simplest (in software) and fastest way to implement the communication link between two control agents. Given enough computer memory for implementation of more complicated profiles, the reader might find a profile suitable for control on layers above L2CAP. As the next section discusses the connection establishment and communication technicalities, it will be more obvious what one requires for a particular application.

2 Link Overview

To the controls engineer, the RF and baseband technicalities are of minor interest. The engineer only needs to go through the connection procedure at the software level and model the connection statistics.

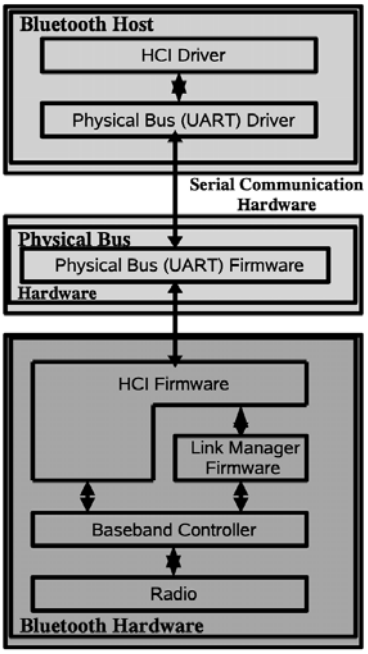


Fig. 4. Accessing hardware via HCI UART

HCI layer

The HCI layer is where software meets hardware in Bluetooth. Let us assume an example scenario where there is a PC with a serial universal asynchronous receiver and transmitter (UART) interface; note that the HCI can also work over USB, PCMCIA, and other interfaces as well, but accessing the HCI layer is always done in a serial-data manner.

Usually a driver deals with HCI, as well as L2CAP and access to the Bluetooth device is only through higher layers. However, a controls engineer might be interested in the HCI layer, not only because it is the basis of software access to the Bluetooth controller but because in some cases it is not possible to have enough software running in the upper layers.

HCI works with command, event, and data packets. Commands are sent to the controller from the host, events are sent in reply to the controller, and data packets are sent both ways. The HCI packet format is given in the specification.

Bluetooth commands are divided into logical groups by function. Some of the group functions include controller configuration and information, device discovery, quality of service, link information, authentication and encryption, and testing.

An essential Bluetooth feature is the ability to scan for remote controllers. An "inquiry" command is issued to the HCI firmware, which sends a beacon signal out to scan for other devices. If there are devices that have "inquiry scanning" enabled[3] they will respond with their Bluetooth device address (BDA). The BDAs will be returned to the host in the form of event packets.

ACL connections

The inquiring host can issue a connection request to the controller if it knows the BDA of the remote device. This is done with a *connection request* command. For commands that take time to complete (like a connection request), there is a *command status* event packet returned from the controller before the command is even complete. The connection requested will be of an ACL type. An ACL connection can be "upgraded" to an SCO connection. Also, when initiating a connection, a device assumes the role of the master. If the "allow role switch" bit is set (in the connection request command), then the connection-accepting device can switch from slave to master. In this case, the device that initiated the connection will end up being the slave.

Bluetooth ACL data is reliable due to the FEC applied on each packet. A number of packet types can be used on each connection. They are specified to the controller when the *create connection* command is issued. Fig. 5 shows the steps (commands and events) involved in successfully establishing a connection. An ACL connection handle is returned in the *connection accepted* packet.

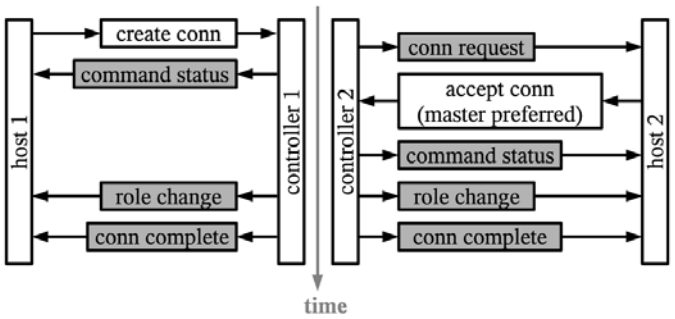


Fig. 5. Commands and events on two sides establishing an ACL connection

By default, the controller that issues the *create connection* command becomes the master. However, referring to the specification, one notices that there is a field in the *create connection* command that allows an automatic role switch. A role switch can also occur later by another explicit command.

[3]Inquiry scanning and page (connection) scanning can be disabled to save bandwidth and power.

3 Control Applications

Bluetooth is very handy when it comes to control applications. The reasons include its low power consumption and the ease of implementation. A Bluetooth ACL connection is not only capable of transmitting switching commands to a controller, it is also very capable of reliably transmitting periodic feedback from a sensor or control commands to an actuator at intervals of the order of milliseconds. From a master controller, one can transmit FEC data packets to more than two client slave controllers with a delay of around 20 ms, a ±5 ms jitter, or at bandwidths approaching 70%-80% of the specification[4]. SCO connections can also be used to transmit data for control. They are used mostly for sound applications (e.g., headsets).

Fig. 6 shows the Ericsson Development Module. It has a USB and a serial UART interface. One can clearly spot the antenna, printed on the right side of the circuit board, and the Ericsson ROK 101 chipset next to it.

Embedded systems

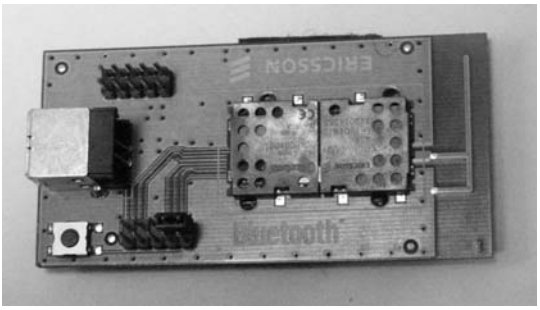


Fig. 6. Ericsson Development Kit

Depending on how customized the control hardware needs to be, an engineer can either purchase a USB-capable microprocessor board or PC and a USB Bluetooth dongle, or build a board using a Bluetooth chipset and microprocessor of choice from scratch. The ROK 101 007 from Ericsson could be a candidate for the Bluetooth chipset on such a board. Other chipsets include the CSR Bluecore [5], the TI BRF6150, and many others. Bluetooth chipsets are either single-chip or multi-chip ICs that have all hardware and firmware below the HCI layer integrated. An antenna and some minor external circuitry are also required.

[4]These are rough measurements that depend on the HCI interface (USB is preferable), the make of modules used, and the environment setting. The numbers are representative, in any case, of the capabilities of the system.

With PC104-size (3.6 x 3.8 in), USB-capable Single Board Computers (SBCs) widely available, the need to build a custom board occurs only in very few, specific cases. A USB Bluetooth 1.1 module prices at around \$30–\$40[5] and a 400–600 MHz SBC costs around \$400–\$500.

The size, as shown on Figure 7, of the USB controllers is quite convenient (compare with the USB connector), not larger than 2 inches in length.

Fig. 7. D-Link^{TM} and Belkin^{TM} USB dongles

In the case where a USB-host-controller[6] is not available, any board with a microprocessor capable of serial communications can be converted or expanded to use Bluetooth. In designing a UART interface, the MAXIM [7] family of chips might be useful.

A variety of companies sell development boards for Bluetooth. These usually include a Bluetooth-integrated, PC-interfaced circuit board and sample source code, along with software drivers. A simple web search on "bluetooth development kit" should return a plethora of sites; for instance, [8] is an excellent example.

3.1 Operating systems and language platforms

Bluetooth source code is widely available throughout the World Wide Web. Most language platforms have had at least one developer compose a Bluetooth software stack for them. There are many Bluetooth implementations in Java, Python, C, and other languages, and the same is true for hardware-specific Bluetooth stacks (microprocessor-specific). Since the interface with HCI is

[5] Second quarter, 2004.

[6] An IC that interfaces with the main processor and masters a USB.

basically serial message passing, and it is the only layer that communicates with hardware, everything above that layer can easily be cross-platform. For controls, a fast non-interpreted language like C is suggested.

All well-known operating systems support Bluetooth, most with a variety of drivers.

An open-source implementation

Bluetooth is an open-stack protocol. Everything is documented by the specification, and any individual or company may implement the protocol in hardware and software.

Linux, being an open-source operating system, has an open-source software implementation of Bluetooth, compatible with almost all devices in the market. The stack officially included with the operating system's source code is the *Bluez* stack.[7]

Bluetooth stacks

By looking into *Bluez* C source code, an engineer can write drivers for any operating system or platform. Therefore, it is the software stack of choice for this article. Documentation on the MS Windows implementation is available on the Microsoft Developers' Network (MSDN) web [10]. The Apple implementation is given in [11]. *Bluez* [9] uses the sockets layer to communicate Bluetooth information back and forth to the user program and drivers. UNIX sockets commands such as `open()`, `socket()`, `write()`, and `recv()` are used with sockets of type `SOCK_SEQPACKET`, protocols `BTPROTO_L2CAP` or `BTPROTO_HCI`, and domain `PAF_BLUETOOTH`.

Connections

One can either establish connections at the L2CAP level or at the HCI layer. Since the interface is available, there is no reason not to use L2CAP, which is a higher, neater layer. HCI-level functionality is still needed for issues such as turning *scan enable* on, resetting the module, etc. There is practically no overhead in processing time for running L2CAP sockets over raw HCI. The L2CAP layer takes care of all the buffering that otherwise would be done by the user program. No software buffering is done in the HCI layer—it is merely the message passing layer which communicates with hardware. There are events that specify how many commands can be accepted by the controller and a command that lets the host specify how large its buffers for receiving data are. This is the only regulation done in HCI. In general, raw HCI should only be used when the socket interface is not available—in systems with small memory, or in systems that do not have or need the Bluetooth stack implemented to the L2CAP level.

[7] There are other software implementations that were developed under Linux, like the Axis stack, which was abandoned after *Bluez* was chosen as the official Linux stack.

Lower level drivers

The HCI drivers *hci_usb.c* and *hci_ldisc.c* in the Linux kernel *drivers/bluetooth* path are a good reference for building USB and UART drivers from scratch.

There might be a case in which the implementation is straightforward and can be hard-coded into a system. No layering is required in that case. An example would be a microcontroller which after booting connects via a Bluetooth controller to a remote host and starts streaming data to it. In this case, the user just needs to run a sequence of commands in series and await the respective packets. The only robustness that can be built in would be based on time-outs or erroneous and unexpected events at each stage of the process. A fall-back then occurs, as shown in Fig. 8.

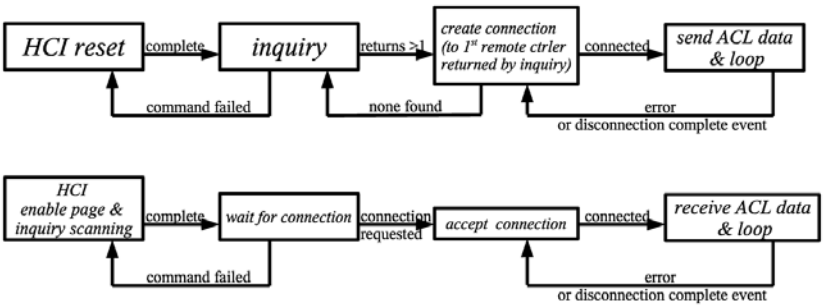


Fig. 8. Fall-back flow for two Bluetooth nodes (master above slave)

A layered approach allows a lower layer to keep track of connections similarly to L2CAP, which lets all higher layers perform in a fire-and-forget mode. If the lower layer, after a number of specified attempts, fails, then the failure carries over to the layer above it, which in the meantime could have been performing other tasks. The layered approach provides more robustness and allows for multi-tasking. This is similar to how the L2CAP layer works.

Known issues

While designing for Bluetooth, one could run into several known issues that may have been tackled in the past by designers. Some are operating-system specific, some are hardware-brand specific, and some are generally applicable to all setups.

- When sending ACL data packets at the HCI level, even though L2CAP may not be implemented, the data packets need to be **L2CAP valid**. That is, the Bluetooth controller will not send ACL packets if they do not have a valid L2CAP header. The following listing shows how a byte stream could be converted to a valid L2CAP ACL packet. See the specification on more details and definitions of the packet headers.

```
#define MAX_ACL_PKT_SIZE 256
#define ACL_PKT 0x02
int hci_send_acl_data(int handle, char *data,int count)
{
  char c[MAX_ACL_PKT_SIZE];
  c[0] = ACL_PKT;
  c[1] = (char)handle;
  c[2] = 0x20;
  c[3] = (count+4) & 0x00ff;
  c[4] = ((count+4) & 0xff00)>>8;
  c[5] = (count) & 0x00ff;
  c[6] = ((count) & 0xff00)>>8;
  c[7] = (char)handle;
  c[8] = 0x00;
  memcpy(&c[9], data, count);
  uart_write(c, count+9);
  return 0;
}
```

- In Bluetooth, the controller that initiates the connection is the piconet's master, unless there is a role switch later on. When a device is active, as mentioned before, it should not be configured to be slave in multiple piconets. When multiple "clients" are initiating connections to a single server, a role switch must occur after each connection so that the server ends up being the only master in the network.
  There are commands such as *read local supported features* that can be issued at the HCI layer and will return the capabilities of the module. In the Linux stack, the *Bluez* utility `hciconfig` will display the supported features. These capabilities include the size of the controller's buffers as well as the number of SCO connections possible.
- To perform role switch under Linux, one can use the `getsockopt()` and `setsockopt()` commands. The bit `L2CAP_LM_MASTER` determines if the local controller will be the master.
- The best way to start a point-to-multipoint server-to-clients network is to connect to all clients first (or have all clients connect to the server and then perform a role switch for each client), and *then* start broadcasting data. Some controllers have trouble connecting more slaves to their piconet if they have already transmitted data to those already connected.
- To get started on writing software for HCI from scratch, try to issue a *reset* command and read back the resulting event from a controller. After that, try to enable inquiry scans and issue an inquiry command, and then read the results. The source code for *Bluez* drivers, libraries, and utilities is one of the best references.
- Some RF specification details:

The range of 10 m for class 2 devices is actually more than specified. On an open corridor, they range to 20 m or more. However, a wall boundary (no door) is enough to stop a connection. Connections may take more than 5 s to establish at any range.
Therefore, if one is seeking to create an interesting ad-hoc network with mobile agents that go out of range, the way to establish out-of-range disconnections is to have the agents switch rooms, not necessarily get far away in line-distance from each other.

- The *Bluez* website has a listing of all hardware [12] that works with the *Bluez* software. Some controllers require their firmware to be updated every time the driver starts. They list "+bluefw" ("fw" for firmware) in this case. The *bluefw* is also open source and available for download at the *Bluez* website.

4 Reference Projects

4.1 The HoTDeC project

The Hovercraft Testbed for Decentralized Control [13] at the University of Illinois utilizes Bluetooth to send commands and vision data from an overhead camera to a group of hovercraft. The vision server, running on a Linux Pentium PC, uses the *Bluez* software and drivers to communicate via a USB Bluetooth controller. There are two versions of hovercraft. The first ran Bluetooth on a TI DSP and the latest version has a Linux SBC running the *Bluez* stack like the server.

The network and information structure

The first version of the HoTDeC vision system generated coordinates x, y, and θ of each of three moving objects, two hovercraft and one free-moving airhockey puck, at 60 Hz. The coordinates of all objects were sent to both hovercraft with a delay of approximately 25 ms. In Bluetooth terms, every 17 ms two slave controllers (hovercraft) in a single piconet received a 50-byte packet (vision data) with a 25 ms delay from the originating host. As mentioned above, the network connections were formed before any vision data was transmitted for the first time to any of the receiving controllers.

Broadcasting

The vision data was transmitted using two methods: unicasting and broadcasting. For every slave in the piconet, the master has a respective ACL connection handle to use when sending or receiving data. Broadcasting to all slaves in the piconet can be done by sending an ACL packet to a handle number that does not correspond to any of the handles for the slaves. No gain in bandwidth or delay was obtained by broadcasting, in the HoTDeC case.

Transport layer issues

The receiving controllers on the hovercraft used UART HCI interfaces, which are 20 times slower[8] than USB. The latency experienced beyond the transport layer remains the same whether HCI is USB or UART. Hence, even though the second version of the HoTDeC network featured USB HCI controllers on the hovercraft, the delay could not be limited below 20 ms.

Slave-to-slave messaging

For hovercraft-to-hovercraft communications, the messaging used was indirect. Due to the fact that the bandwidth required for this was minimal, there was no need to form more than one piconet: the messages from hovercraft to hovercraft were relayed through the vision server's controller.

Contacting the designers

HoTDeC was conceived and designed at Urbana, Illinois, by a group including the authors. The source code is not open, but can be provided free of charge and reused at no charge or warranty, if requested, for research purposes.

4.2 Tinyphoon

A commercial application, *tinyphoon* [14], incorporates Bluetooth in autonomous wheeled robots that play robotic soccer.

4.3 Lund Institute of Technology

Research at Lund, Sweden, documented in Andreas Hörjel's Masters thesis [15] and [16] demonstrates and deals with delays and other issues in using Bluetooth for control. Also, [17] is an excellent reference. The Harald Stack [18] and JSR-000082 APIs [19] use Java for Bluetooth.

5 Finding More Information

Subscribing to the *Bluez* developer and user mail lists is suggested. The authors of the *Bluez* software are very helpful and answer users' questions, through e-mail lists, almost in real time. Archives of the mailing list also provide previous posts and answers to frequently asked questions. Since webpages are often removed, web-links might become invalid after a few years. The best way to look for information is to use keywords and read newsgroups. Some useful keywords/phrases that may not be obvious are: *"bluez-devel", "bluetooth dongle", "bluetooth stack", "development kit", "bluetooth chipset", "bluetooth single-chip",* etc.

[8] 56kbps for UART versus 1.1Mbps for USB 1.0.

Contacting the authors

Please feel free to e-mail `vladimer@uiuc.edu` for feedback or directions on starting a Bluetooth project.

References

1. The Bluetooth Special Interest Group trade association `http://www.bluetooth.com/about/`
2. The Bluetooth Core Specification v1.2 can be downloaded from `https://www.bluetooth.org/spec/` in portable document format.
3. James Kardach, Mobile Computing Group, Intel Corporation, Intel Technology Journal 2nd quarter 2000, Bluetooth Architecture Overview (`http://www.intel.com/technology/itj/q22000/articles/art_1.htm`)
4. `http://www.palowireless.com/bluetooth/`
5. The Bluecore family of products, at the Cambridge Silicon Radio website: `http://www.csr.com/products/bc-family.htm`
6. The PC104 and PC104+ standard website `http://www.pc104.org/`
7. Dallas Semiconductor MAXIM chips `http://www.maxim-ic.com/`
8. Palo-wireless listing for Bluetooth development tools: `http://www.palowireless.com/bluetooth/devtools.asp`
9. Bluez Official Linux Bluetooth protocol stack: `http://www.bluez.org`
10. MSDN website for the Windows™ Bluetooth stack: `http://msdn.microsoft.com/library/default.asp?url=/library/en-us/wcebluet/html/ceconBluetoothStackArchitecture.asp`
11. Developer Connection for Apple™ Bluetooth `http://developer.apple.com/hardware/bluetooth/`
12. Information on Bluetooth hardware, compatible devices, and other topics on the *Bluez* website: `http://www.bluez.org/hardware.html`
13. The HoTDeC project webpage: `http://legend.me.uiuc.edu/hotdec`
14. Tinyphoon webpage: `http://www.tinyphoon.com`
15. A. Hörjel, Bluetooth in Control, Masters thesis, ISRN LUTFD2/TFRT–5659–SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, January 2001.
16. J. Eker, A. Cervin, and A. Hörjel, Distributed wireless control using bluetooth, *IFAC Conference on New Technologies for Computer Control*, Hong Kong, P.R. China, November 2001.
17. M. Attnäs and U. Laurén, Porting the Ericsson Bluetooth stack—A real-time analysis, Masters thesis, ISRN LUTFD2/TFRT–5684–SE, Department of Automatic Control, Lund Institute of Technology, Lund, Sweden, 2002.
18. The Harald Bluetooth stack in Java: `http://www.control.lth.se/~johane/harald/`
19. The JSR-000082 API: `http://developers.sun.com/techtopics/mobility/midp/articles/bluetooth2/`
    Download at: `http://jcp.org/aboutJava/communityprocess/final/jsr082/index.html`
20. R. Antony and B. Hopkins, Bluetooth for Java, ISBN: 1-59059-078-3, Mar 2003. Info at: `http://www.javabluetooth.com/`

The Cornell RoboCup Robot Soccer Team: 1999–2003

Raffaello D'Andrea

Sibley School of Mechanical and Aerospace Engineering,
Cornell University, Ithaca, NY 14853, U.S.A.
raff.d'andrea@cornell.edu

1 Introduction

What is RoboCup? Taken directly from the RoboCup Organization web site (www.robocup.org):

> RoboCup is an international research and education initiative. It is an attempt to foster AI and intelligent robotics research by providing a standard problem where a wide range of technologies can be integrated and examined, as well as being used for integrated project-oriented education.

While the scope of RoboCup has grown since its inception in 1997, the main activity remains the robot soccer leagues: Simulation, Small Size, Middle Size, Four-legged, and Humanoid. The Simulation league, as the name implies, does not involve physical hardware, except for computer workstations. The Four-legged league consists of Sony Aibos—robot dogs—programmed to play soccer. The Humanoid league is still in its infancy; the main objective of this league is to demonstrate basic skills, such as standing, walking, and kicking a ball, with a two-legged robot. The remaining two leagues, the Small Size and Middle Size leagues, involve the actual construction of robots that compete in head-to-head soccer matches.

The main difference between the two leagues, contrary to what the names suggest, is the allowed sensing technologies. Teams in the Middle Size league are constrained to use local, on-board vision, while competitors in the Small Size league are permitted to use global vision systems (in addition to local vision systems, if they so desire). As a result, the Middle Size league is dominated by robot localization and perception, while the main challenges in the Small Size league are at the system and integration level. In addition, due to the enhanced situational awareness obtained with global vision, the Small Size league games are much faster and typically involve more team play and collaboration, such as passing.

This chapter describes the Small Size league Cornell RoboCup team in the time period starting in 1999, the year it first competed, and ending in 2003, after five years of competition. In this time period, the Cornell team won the competition four times and placed third once. See Table 1 for a summary of the competition outcomes since inception or the RoboCup web site for a more detailed summary. The emphases of this chapter are on the parts of the system that distinguished the Cornell team from its competitors and led to its success in competition.

In the Small Size league the soccer matches are played by teams composed of up to five robots. Similar to the real game of soccer, the objective is to score more goals than the opponent, subject to well-defined rules and regulations. For example, repeated pushing and charging fouls lead to yellow cards (warnings) and eventual red cards (expulsions). The reader is referred to the RoboCup web site for a detailed list of the rules. The robots and an off-field computer are permitted to communicate via wireless transceivers. A global vision system is typically used for robot and ball position determination. Fig. 1 is a picture from a typical game.

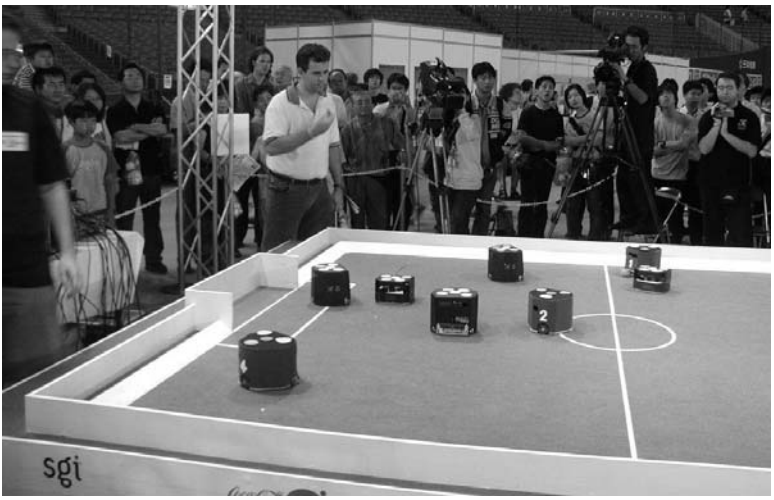


Fig. 1. A typical RoboCup game, with the human referee getting ready to blow the whistle and initiate game play

RoboCup is an excellent test bed for developing new tools and techniques for controlling autonomous systems in uncertain and dynamic environments. From an educational perspective, it is also a great means for exposing students to the systems engineering approach for designing, building, managing, and maintaining complex systems [1], [2]. In addition to the RoboCup web site, the reader is referred to [3], [4], [5], [6], and the references therein, for an in-

depth description of the competition, its history, and the long-term objectives of RoboCup.

Year	Location	Teams competing	Champion
1997	Nagoya, Japan	4	Carnegie Mellon University
1998	Paris, France	12	Carnegie Mellon University
1999	Stockholm, Sweden	18	Cornell University
2000	Melbourne, Australia	20*	Cornell University
2001	Seattle, USA	20*	Nee Ann Polytechnic
2002	Fukuoka, Japan	20*	Cornell University
2003	Padova, Italy	20*	Cornell University

Table 1. Small Size league results.
*Since 2000, the competition has been restricted to the top 20 teams, as judged by qualification papers and videos.

2 Physical System Architecture

The physical architecture used by the Cornell team is depicted in Fig. 2. A global vision system, with up to three cameras, was used to identify game objects. This information was then fed to the control workstation, whose task was to coordinate the motion of the robots. Various versions of the Windows operating system were used. A low-latency version of transmission control protocol (TCP)/Internet protocol (IP) was used for inter-workstation communication. The total round-trip latency of the system was different every year. It constantly decreased as new technology was introduced and better algorithms were developed. In 2003, the total system latency was approximately 80 ms.

3 The Robots

A picture and CAD drawing of a 2003 robot are shown in Fig. 3. Functionally, each robot had the following components: a drive mechanism for locomotion; a dribbling mechanism for imparting back-spin on the ball; a solenoid for kicking and passing the ball; and microprocessor-based electronics for motor control, kick and dribble control, and wireless communication. Some of these are described below.

3.1 Drive mechanism

Cornell introduced the omni-directional drive for locomotion in 2000, an innovation which was adopted by most other teams. For example, in the 2003

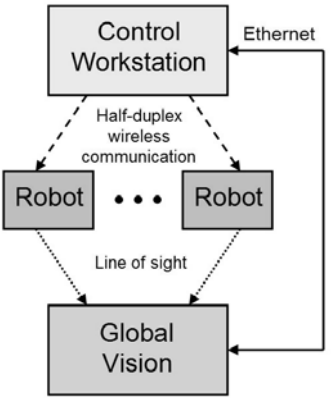


Fig. 2. Physical architecture

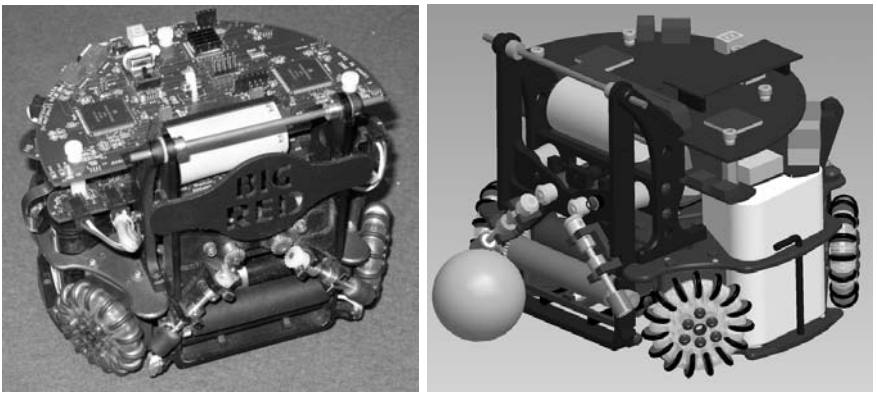


Fig. 3. 2003 robot

competition, seven out of eight teams in the quarter finals were using omnidirectional drives. There are two main advantages for using these drives over their standard two-wheeled counterparts: 1) the robots are more maneuverable, and 2) the problem of trajectory generation is substantially simpler [7]. Contrary to common belief, it is the second advantage that makes omnidirectional drive much more superior than two-wheel drive, when viewed from an *overall system* perspective. This is discussed in more detail in Section 4.3.

The maximum *controlled*[1] speeds and accelerations of the robots are listed in Table 2. A rate gyro was introduced in 2002 to substantially increase the performance envelope of the robots.

[1]The word "controlled" is taken to loosely mean ability to follow a pre-specified trajectory.

Year	Max. Speed (m/s)	Max. Accel. (m/s^2)
1999	1.0	2.0
2000	0.6	1.5
2001	1.0	2.0
2002	1.4	2.5
2003	1.8	3.5

Table 2. Evolution of robot performance. Two-wheeled robots were used in 1999, three-wheeled omni-directional robots in 2000 and 2001, four-wheeled omni-directional robots in 2002 and 2003.

3.2 On-Board electronics

A different microprocessor was used each year of the competition. The choice was essentially dictated by the expertise of the current team members, and the desire to shift more intelligence onto the robot and away from the control workstation. For example, in 2003, various coordinate transformations and some rudimentary prediction and estimation, made possible by the rate gyro, were implemented directly on the robot.

Wireless communication also varied from year to year. For example, dedicated, low-latency wireless modules in the 433, 418, 869, and 914 MHz bands were used in 2002 and 2003. These wireless modules were used in half-duplex mode, with one transmitter (connected to the control workstation, see Fig. 2) and five receivers (on the robots). In 2003, information was sent to the robots at a rate of 60 times per second, although very little performance degradation was observed at the lower rate of 30 times per second.

4 Functional Architecture

The functional architecture is depicted in Fig. 4. There are five main blocks, described in detail below.

4.1 Global vision

The global vision system was responsible for extracting the position and orientation of the friendly robots and position of opponents and the ball. While it was possible to extract the orientation of the opponents, this information was of little use due to the widespread use of omni-directional vehicles and the high rotation speed of two-wheeled vehicles.

Unlike the Middle Size League, computer vision in the Small Size league is relatively straightforward. In particular, all motion is essentially two dimensional (an overhead camera is typically used), and distinct color markers are placed on the tops of the robots. A top view of the 2002 Cornell robots and their vision markers is shown in Fig. 5, as well as the 2002 Vision Calibration Graphical User Interface (GUI).

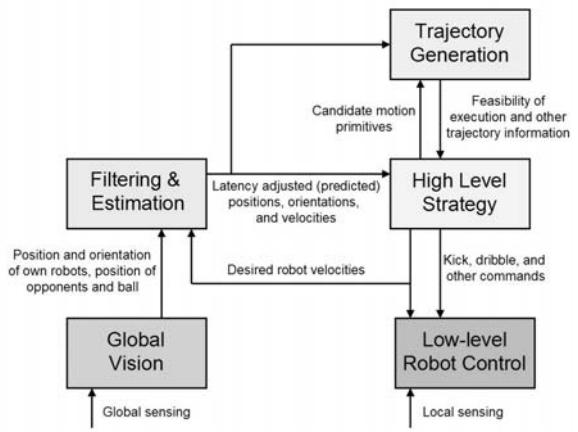


Fig. 4. Functional architecture

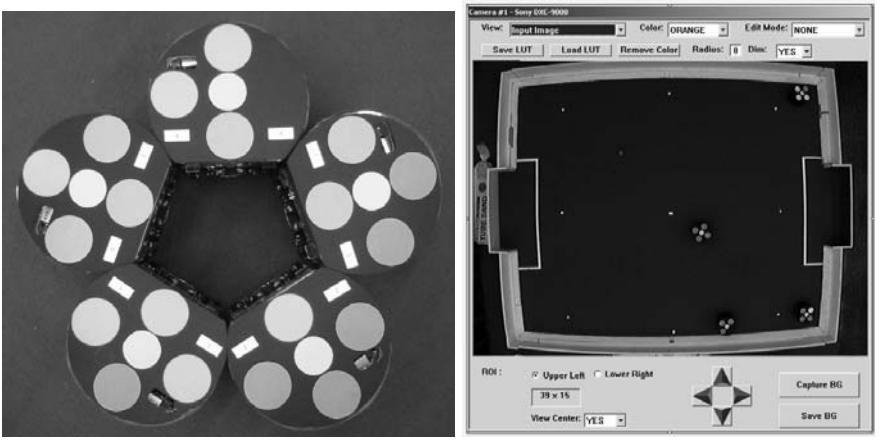


Fig. 5. LEFT: Top view of the 2002 robots. Various color coding schemes were used to make position and orientation determination robust to missed markers. RIGHT: The Vision Calibration GUI. The vision system had to be recalibrated whenever the lighting conditions changed substantially

4.2 Filtering and estimation

This block was responsible for taking the vision data, extracting velocity information from it, and predicting the location of the robots and the ball in the *robot temporal frame of reference*. In particular, as far as all decision-making blocks were concerned, "now" corresponded to the time when a robot was *observed* to act upon a command. For the opponents and the ball, filtering and estimation/prediction were performed using straightforward Kalman filters.

For friendly robots, the commanded robot velocities were also used to obtain a much more accurate prediction of their position. In the absence of disturbances, this essentially nullified the effects of latency. Note that this approach is very similar in spirit to the standard Smith predictor [8], which is often used for time delay compensation.

Various strategies were used to reset the filters when the ball and robots abruptly changed direction (mainly due to collisions), or when substantial discrepancies were observed between the predicted values of the friendly robots and the actual values obtained via computer vision.

4.3 Trajectory generation

The responsibility of this block was to take as inputs candidate motion primitives, such as "Move to location (X,Y), as quickly as possible, without hitting anything", and generating the appropriate robot velocities and other trajectory parameters (time to execute a maneuver, for example).

In the absence of obstacles, nearly time-optimal trajectories were generated for the omni-directional vehicles with very little computational burden: Less than 300 floating-point operations; see [7] for details. A typical trajectory is depicted in the left part of Fig. 6. The basic idea is captured in the right part of Fig. 6. A compensator inserted in the vehicle control loop altered the effective dynamics in a way that greatly simplified the associated optimal control problem for trajectory generation. This simplification was achieved with a very small sacrifice in performance [7].

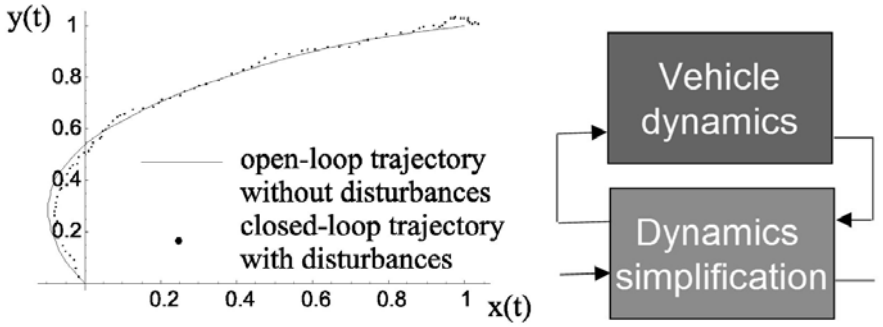


Fig. 6. LEFT: Typical result of trajectory generation. RIGHT: Dynamics simplification

The resulting trajectories were then used as primitives for various obstacle avoidance algorithms, such as the randomized path planning algorithms in [9].

Using standard computing platforms, one could thus perform more than one million trajectory calculations per second. Referring to the diagram of

Fig. 4, the implications should be clear: the High Level Strategy block was thus free to explore a very large number of nearly optimal motion scenarios.

4.4 Low level robot control

This block, which resided on the robots, essentially performed velocity tracking and supervised the control of all other mechanisms on the robot, such as dribbling and kicking. As an example, the block diagram of the inner loop used for angular velocity regulation is given in Fig. 7.

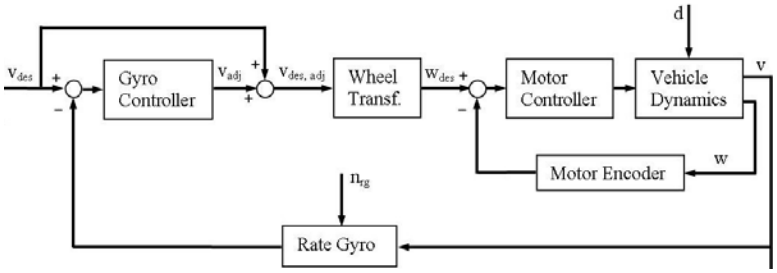


Fig. 7. Angular velocity regulation

4.5 High level strategy

Not surprisingly, this block relied the most on human input for design (such as hard-coded successful sport strategies[2] or scouting the opponents and altering the behavior of the system before game time) and it was the least amenable to systematic design and analysis. This, of course, is not to say that it is a trivial task to design, test, and implement it. Rather, the system was architected so that all the hierarchical levels below it had solid analytical backbones (such as optimal control for trajectory generation and Kalman filtering for estimation), while most of the heuristics were confined to the High Level Strategy block. This block also changed the most from year to year, as we tried new methodologies and approaches.

In 1999, our first year of competition, we adopted a role-based system similar to what was used by the defending 1998 champion, Carnegie Mellon University [10]. This approach was successful and yielded acceptable performance, but it was very difficult to debug and build upon. In particular, role

[2]Interestingly, soccer did not prove to be the best inspiration for successful robot soccer strategies. In many ways, the robotic game is much more like hockey and basketball than soccer.

interdependencies made systematic design extremely difficult; it was often impossible to predict the effects of adding a new role to the system. Having said that, this variability and unpredictability could have its benefits if harnessed by evolutionary optimization techniques.

Fig. 8 captures a simplified version of the High Level Strategy used in 2003. The directed graph in the left part of the figure captures the possible Game States and the transitions between them. One possible Game State, "Opportunistic Offense", is depicted in the right part of the figure. In a given Game State, each robot was assigned a role. Unlike a pure role-based system, the five roles for each Game State were fixed. Each role, in turn, could make use of various skills. Roles could be reused, as could skills. Both roles and skills were typically parameterized, which resulted in small and manageable sets of roles and skills.

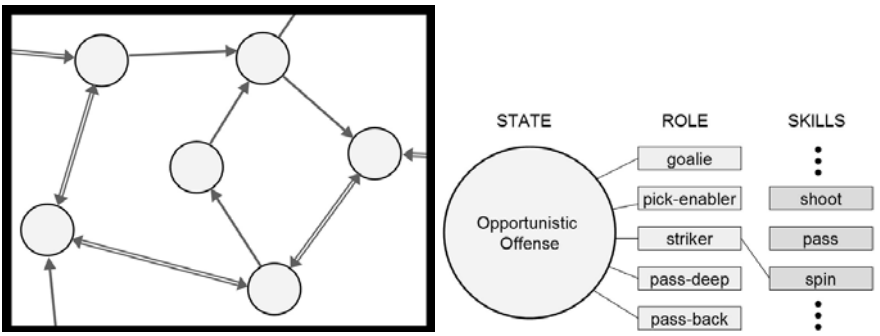


Fig. 8. LEFT: Directed graph with Game States. RIGHT:Assigning roles and skills

One can then think of High Level Strategy in the following terms:

1. When should one switch to a different Game State?
2. How should the roles be assigned among the robots?
3. What skill should a robot be invoking?

The transitions between Game States were in general not deterministic. In particular, randomness was introduced, especially on offense, to allow the exploration, and exploitation, of an opponent's weaknesses; reinforcement learning could then be used to alter the transitional probabilities based on various performance metrics, such as the position of the ball, scoring opportunities, etc. State overlap and hysteresis were used to prevent rapid transitions between Game States. An alternate approach, which was not pursued but is clearly of interest, is to design the system such that Game State transitions are "smooth" and thus do not require overlap and hysteresis. From a conceptual standpoint, this seems much more natural; in most situations, it does not make sense for the behavior of the system to change dramatically when a

transition from one Game State to the next is made. For others, however, it certainly does make sense to have discontinuities; a free kick, for example.

Roles were generally assigned based on a cost function specific to the current Game State, which was in general not only a function of the current state of the system, but also the predicted (future) state of the system. A greedy algorithm, based on the relative importance of the roles, was found to yield performance similar to an exhaustive search. An interesting direction that was not pursued was to extract the relative ordering required for a greedy search from repeated runs of exhaustive searches.

The decision of which skill a robot should use was similarly based on a local cost function, which was in general a function of the future, predicted state of the system. This is, in fact, a simplified version of what was actually implemented. Skill selection could actually be coordinated among the robots. It was observed, however, that skill coordination was difficult to implement and only helped in limited circumstances.

High Level Strategy development was greatly aided through the use of a dedicated simulator, such as the one depicted in Fig. 9. The main use of the simulator was for designing the directed graph of Game States and for designing roles. Skill development, however, was typically performed directly with the robots. It was thus a design requirement that the robot skills satisfied the specifications used in the simulator, or conversely, that robot skills were faithfully captured by the simulator.

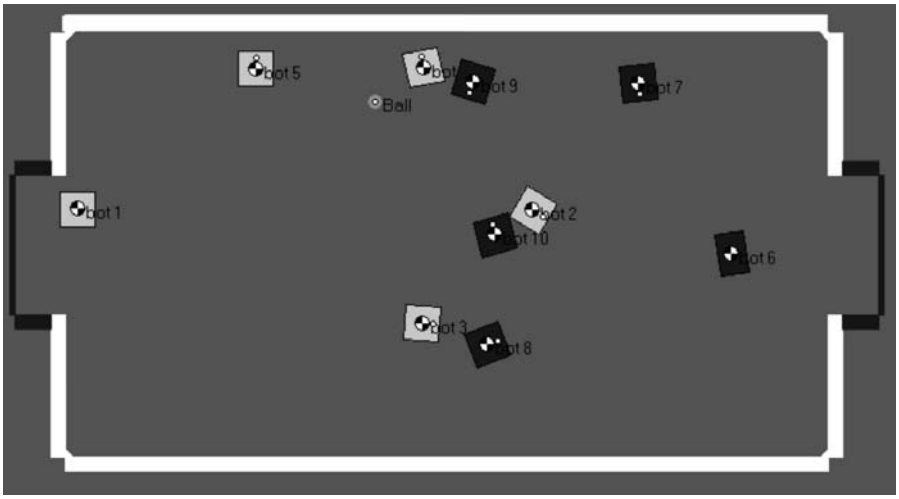


Fig. 9. Screen shot of first simulated RoboCup game, played in February 1999. The simulation engine was Working Model 2D, which was interfaced to MATLAB for real-time control.

5 Lessons Learned

The underlying system architecture is crucial to the success of a team. In particular, it should be modular and easy to adapt, and should allow a large number of individuals to concurrently develop the system. This modularity should appear at all system levels: from the high level decision makers down to the physical robots. While this may sound obvious, flexibility, adaptability, and modularity are often sacrificed for *potential* performance gains that are never realized.

Related to this point is the appropriate use of abstractions, again at all system levels. For example, the use of motion primitives insulates the High Level Strategy block from the details and complexities of robot motion control, which allows individuals with very little knowledge of robotics to design successful high level strategies.

A solid understanding of feedback, dynamics, and control is critical. For example, feedback is an excellent tool for making subsystems modular and thus easier to use as building blocks by non-experts. The appropriate use of filtering, estimation, and prediction can greatly simplify the interface between the robots and the high level algorithms that ultimately control them.

The final point is related to the objectives of the competition itself. Many teams use the RoboCup competition as a testbed for "artificial intelligence" and machine learning. In order to do well at the competition, however, it was much more useful to design a system that readily accepted inputs from humans and was readily amenable to redesign than to develop a system that "learned" on its own. One of the keys to our success in the RoboCup competition was our ability to very quickly encode human information into the system, such as implementing new strategies based on the scouting of opponents.

Not only was this important for winning the competition, it is also much more technologically relevant. There is nothing that currently comes close to the problem-solving abilities of humans, especially in environments where decisions must be made quickly. It thus makes sense to push research in a direction that utilizes these human assets, rather in one that marginalizes them. By doing so, we will gradually, but incessantly, reduce the amount of human input required to supervise and operate these multi-asset systems, and move that much closer to building truly "intelligent" systems.

References

1. R. D'Andrea. Robot soccer: A platform for systems engineering. *Computers in Education Journal*, 10(1):57–61, 2000.
2. M. Asada, R. D'Andrea, A. Birk, H. Kitano, and M. Veloso. Robotics in edutainment. In *IEEE International Conference on Robotics and Automation*, pages 795–800, 2000.
3. R. D'Andrea and J. W. Lee. Small size league winner: Cornell Big Red. *AI Magazine*, 21(3):41–44, 2000.

4. P. Stone, M. Asada, T. Balch, R. D'Andrea, M. Fujita, B. Hengst, G. Kraetzschmar, P. Lima, N. Lau, H. Lund, D. Polani, P. Scerri, S. Tadokoro, T. Weigel, and G. Wyeth. RoboCup-2000: The fourth robotic soccer world championships. *AI Magazine*, 22(1):11–38, 2001.
5. P. Stone, T. Balch, and G. Kraetzschmar, editors. *RoboCup-00: Robot Soccer World Cup IV*. Lecture Notes in Computer Science. Springer, 2001.
6. R. D'Andrea, P. Ganguly, T. Nagy, and M. Babish. The Cornell robot soccer team. In P. Stone, T. Balch, and G. Kraetzschmar, editors, *RoboCup-00: Robot Soccer World Cup IV*, Lecture Notes in Computer Science. Springer, 2001.
7. T. K. Nagy, R. D'Andrea, and P. Ganguly. Near-optimal dynamic trajectory generation and control of an omnidirectional vehicle. *Robotics and Autonomous Systems*, 46:47–64, 2004.
8. O. J. M. Smith. Closer control of loops with dead time. *Chem. Eng. Progress*, 53(5):217–219, 1957.
9. E. Frazzoli, M. A. Dahleh, and E. Feron. Real-time motion planning for agile autonomous vehicles. *AIAA Journal of Guidance, Control, and Dynamics*, 25(1):116–129, 2002.
10. M. Asada and H. Kitano, editors. *RoboCup-98: Robot Soccer World Cup II*. Lecture Notes in Computer Science. Springer, 1999.

Index