MAGIC
(Multi Target Graphical User Interface)

Pankaj P. Babhare ,Khushbu W. Patre
VIII Semester Dept. of Information Technology
Vilasrao Deshmukh College of Engineering , Mouda, Nagpur
India

pankajbabhare010@gmail.com 20khushbu13@gmail.com

9403348837 8551831843

Abstract
Our goal is to design a better, simple and easy to understand graphical interface description language (MAGIC Script). The script has to be converted to the target languages as per the need of the user; making the compiler Multi-Target. The system consists of a compiler for compiling and translating MAGIC script into specified target language, which may be Java Swing, Java AWT, XUL or C#. The system must be expandable for inclusion of new languages. An IDE for developing the MAGIC Script is to be provided. First the end user will submit the Graphical user interface designed using the MAGIC Script as the input to the system i.e. MAGIC Compiler. Next, he will select the target language in which he wants the code to be converted. The system in the meantime will generate tokens and parse them for finding token and syntax errors in the MAGIC Script submitted and report any errors to the user. From the target language selection the MAGIC Compiler will generate the code in the specified language.

key words- Magic, scope, working, prototype, design, testing.

I. INTRODUCTION

Magic was made to be Multi-Target for most IDEs generate a GUI in a single Language; making the task of converting them to other languages difficult. MAGIC Script is convertible to many well-known and much used GUI definition languages.

To work with MAGIC, the most important decision was to decide on the language for the compiler to compile MAGIC SCRIPT. After many deliberations we decided on using JAVA as the compiler Language.

To write the Parser we have used JavaCC. JavaCC is a parser generator and a lexical analyzer generator. Parsers and lexical analyzers are software components for dealing with input of character sequences. Compilers and Interpreters incorporate lexical analyzers and parsers to decipher files containing programs. It is a tool that reads a grammar specification and converts it to a Java program that can recognize matches to the grammar. Another important task was to select target languages for MAGIC. We zeroed on Java Swing, Java AWT, and XUL and C #.NET. Type Style and Fonts.

II. SCOPE

Design and develop a scripting language which we are calling as MAGIC script Design syntax and semantics for the MAGIC scripting language. Develop a compiler which will be able to understand and execute the MAGIC script written by the developers of MAGIC Develop a compiler component (can be called as a translator) which will be able to translate the code written in MAGIC script to various implementation languages supported by MAGIC.

Design a IDE which can be used by the developers of the MAGIC script, it will assist in developing the user interface code in MAGIC and will also be able to compiler, translate and execute the script written.

III. SYSTEM DESCRIPTION

MAGIC is a script which more or less resembles a modular design of the graphical user interface. The system has an IDE where the developers of MAGIC will be writing the code/programs. The IDE is with facilities which will assist the users in developing the code, such as syntax colorizing, inline compiling options, output console window etc.

The users will be writing a code for graphical user interface and will compile it with the MAGIC compiler. MAGIC compiler is the heart of the system and is used for compiling, executing and translating the MAGIC code.

The final output can be seen with the MAGIC IDE, also according to the user options, the code written in MAGIC script can be translated to various languages supported by MAGIC environment. One code written in MAGIC script can be translated to various implementation languages without the need of rewriting. This will help the developers of the graphical user interface to save time and effort

IV. WORKING OF THE SYSTEM

The main component of the system here is the MAGIC script and its syntax. We here are presenting some of the major components of the graphical user interface

development and are providing the MAGIC scripts for the same. Actual working of MAGIC (Multi target Graphical Interface Compiler) is as shown below.

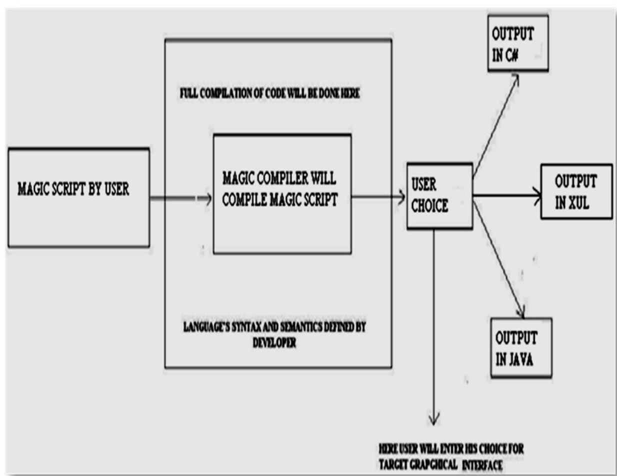

Fig 1*:* Magic compiler

Our system works in the following manner:

First the end user will submit the Graphical user interface designed using the MAGIC Script as the input to the system i.e. MAGIC Compiler. Next, he will select the target language in which he wants the code to be converted.

The system in the meantime will generate tokens and parse them for finding token and syntax errors in the MAGIC Script submitted and report any errors to the user. From the target language selection the MAGIC Compiler will generate the code in the specified language.

A. FEASIBILITY STUDY

Feasibility means practical implementation and practical usage. Our MAGIC is having such a design that is scalable in nature that means many other languages can be added to it. We have already implemented the units for generating target languages-AWT, Swing; XUL and C#.NET .Other languages can be added as per usersø requirements.

MAGIC can be easily used for small to medium sized projects. Its usability for beginners is particularly high.

B. TECHNICAL FEASIBILITY

The development of MAGIC requires very basic resources like Java2sdk 1.4.1, JavaCC parser, XUL Runner and Visual studio with .NET framework. In absence of Visual Studio any C# running software may be used. MAGIC can run on most OS as java is supported by all OSs. So, there are no technical feasibility issues as far as development and implementation of MAGIC is concerned.

As such MAGIC only requires Java Runtime Environment to be installed at user site. It doesnøt require any other software or DLLs to run. Also no other special hardware requirements are there. Hence is very much technically feasible for the user.

C. ECONOMIC FEASIBILITY

MAGIC doesnøt require any costly hardware or software. All the software used is freely available. At user site also there is no requirement of any software only JRE is required.

V. PROJECT PROCESS MODEL

Development occurs as a succession of releases with increasing functionality. Testing and feedback on each release is used in deciding requirements and improvements for next release. There is no õmaintenanceö phase ó each version includes both problem fixes as well as new features. This may also include õre-engineeringö ó changing the design and implementation of existing functionality, for easier maintainability.

We have followed a spiral model like approach for our project as it has a large no of units. So, prototyping at each stage is necessary. The spiral model is a software development process combining elements of both design and prototyping-in-stages, in an effort to combine advantages of top-down and bottom-up concepts. Also known as the spiral lifecycle model, it is a systems development method (SDM) used in information technology (IT). This model of development combines the features of the prototyping model and the waterfall model. The spiral model is intended for large, expensive and complicated projects.

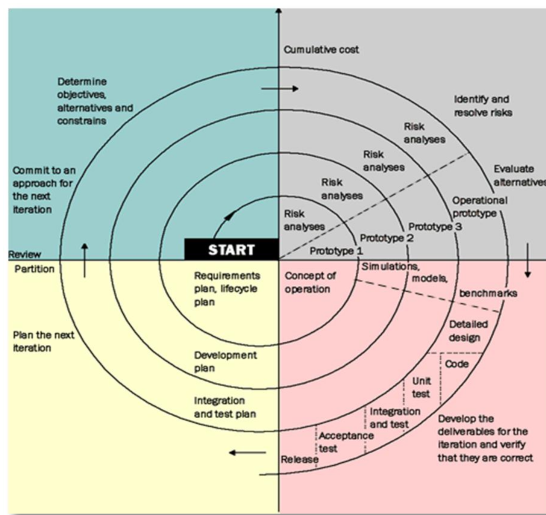

Fig 2: Spiral Model

VI. MAGIC PROTOTYPES

Prototype 1:

a) No support for layouts in frames, windows or panels
b) No deterministic statements, only followed what was given in definitions
c) No sharing of GUI properties, only refer their own properties provided
d) Simple but good code generation translates MAGIC into Java files
e) No compiler output messages informing users what's happening
f) No simple way of adding new components by editing very few areas of definitions file
g) Support for: frames, panels, menus, menuItems, textfields, textareas, buttons, labels and windows
h) Need to sort out these problems by rewriting the definitions file
i) No support for action listeners in MAGIC
j) Future support for scrollbars, actions and other features that are non-GUI related

Prototype 2:

a) Rewritten Java files and definitions file
b) Support for layouts in frames, windows and panels
c) Compiler messages exist now, needs a few adjustments
d) Same code generation, needs some improvements
e) Sharing of GUI properties by the use of Java inheritance
f) Simple addition of new components by editing very few areas of definitions file
g) Dynamic checking and solving of properties of components
h) Nothing done on future support as of yet
i) Actions will be present in next prototype

Prototype 3:

a) Final JavaCC definitions are written to create the new compiler
b) Now contains support for procedures, integers, strings, assignment, calling of procedures, returning values
c) Now contains support for main GUI components and their properties changing ability, setting properties initially
d) Now has support for actions
e) Now has support to target other components within a specific action
f) Support for other actions will be present in next prototype
g) Better error reporting facilities - quoting line, column numbers, error reasons and also the number of errors found
h) Error reporting is not finished and the full scale error reporting will be present in prototype 4

Prototype 4:

a) Added support for ChangeListener which is from Swing Event Model
b) Partial completion of error reporting, only done in definitions file
c) Errors report full information that is helpful to the user
d) New GUI components added, listboxes, comboboxes, popup menus, sliders
e) JWindow bugs discovered, fixed fully

VII. SYSTEM DESIGN

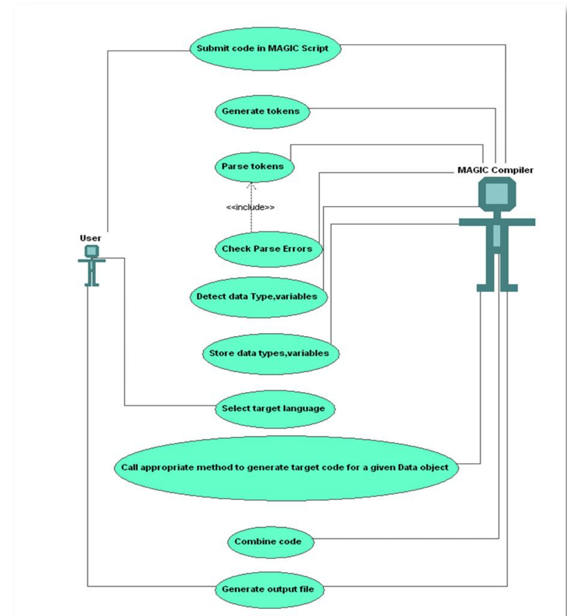

Fig3: Use Case Diagram

VIII. COMPILER CONSTRUCTION FUNDAMENTALS

Programming languages are often divided, somewhat artificially, into compiled and interpreted languages, although the boundaries have become blurred. The concepts discussed here apply equally well to compile as well as interpreted languages.

Compilers have to perform three major tasks when presented with a program text (source code):

1. Lexical analysis
2. Syntactic analysis
3. Code generation or execution

The bulk of the compiler's work centers on steps 1 and 2, which involve understanding the program source code and ensuring its syntactical correctness. We call that process *parsing*, which is the *parser's* responsibility.

Lexical analysis (lexing)

Lexical analysis takes a cursory look at the program source code and divides it into proper *tokens.* A token is a significant piece of a program's source code. Token examples include keywords, punctuation, literals such as numbers, and strings. Nontokens include white space, which is often ignored but used to separate tokens, and comments.

Syntactic analysis (parsing)

During syntactic analysis, a parser extracts meaning from the program source code by ensuring the program's syntactical correctness and by building an internal representation of the program.

Code generation or execution

Once the parser successfully parses the program without error, it exists in an internal representation that is easy to process by the compiler. It is now relatively easy to generate machine code (or Java bytecode for that matter) from the internal representation or to execute the internal representation directly. If we do the former, we are compiling; in the latter case, we talk about interpreting.

JavaCC

JavaCC, available for free, is a parser generator. It provides a Java language extension for specifying a programming language's grammar. JavaCC was developed initially by Sun Microsystems, but it's now maintained by MetaMata. Like any decent programming tool, JavaCC was actually used to specify the grammar of the JavaCC input format.

Moreover, JavaCC allows us to define grammars in a fashion similar to EBNF, making it easy to translate EBNF grammars into the JavaCC format. Further, JavaCC is the most popular parser generator for Java, with a host of predefined JavaCC grammars available to use as a starting point.

JavaCC: Grammar Files

This section contains the complete syntax of Java Compiler grammar files with explanations of each construct.

Tokens in the grammar files follow the same conventions as for the Java programming language. Hence identifiers, strings, characters, etc. used in the grammars are the same as Java identifiers, Java strings, Java characters, etc.

White space in the grammar files also follows the same conventions as for the Java programming language. This includes the syntax for comments. Most comments present in the grammar files are generated into the generated parser/lexical analyzer.

Grammar files are preprocessed for Unicode escapes just as Java files are (i.e., occurrences of strings such as \uxxxx - where xxxx is a hex value - are converted to the corresponding Unicode character before lexical analysis).

Exceptions to the above rules: The Java operators "<<", ">>", ">>>", "<<=", ">>=", and ">>>=" are left out of JavaCCøs input token list in order to allow convenient nested use of token specifications. Finally, the following are the additional reserved words in the Java Compiler Compiler grammar files.

EOF	IGNORE_CASE	JAVACODE	LOOKAHEAD
MORE	PARSER_BEGIN	PARSER_END	SKIP
SPECIAL_TOKEN	TOKEN	TOKEN_MGR_DECLS	

Any Java entities used in the grammar rules that follow appear italicized with the prefix *java_* (*e.g.*, *java_compilation_unit*).

javacc_input ::= javacc_options
"PARSER_BEGIN" "(" <IDENTIFIER> ")"
java_compilation_unit
"PARSER_END" "(" <IDENTIFIER> ")"
(production)*

PARSER_END(parser_name)

JavaCC does not perform detailed checks on the compilation unit, so it is possible for a grammar file to pass through JavaCC and generate Java files that produce errors when they are compiled.

If the compilation unit includes a package declaration, this is included in all the generated files. If the compilation unit includes imports declarations, this is included in the generated parser and token manager files.

The generated parser file contains everything in the compilation unit and, in addition, contains the generated parser code that is included at the end of the parser class. For the above example, the generated parser will look like:

```
. . .
class parser_name . . . {
  . . .
  // generated parser is inserted here.
}
. . .
```

javacode_production ::= "JAVACODE"
java_access_modifier
java_return_type *java_identifier*
"(" *java_parameter_list* ")"
java_block

The JAVACODE production is a way to write Java code for some productions instead of the usual EBNF expansion.

bnf_production ::= *java_access_modifier java_return_type java_identifier* "(" *java_parameter_list* ")" ":"
java_block
"{" expansion_choices "}"

The BNF production is the standard production used in specifying JavaCC grammars. Each BNF production has a left hand side which is a non-terminal specification. The BNF production then defines this non-terminal in terms of BNF expansions on the right hand side. The non-terminal is written

exactly like a declared Java method. Since each non-terminal is translated into a method
in the generated parser, this style of writing the non-terminal makes this association obvious. The name of the non-terminal is the name of the method, and the parameters and return value declared are the means to pass values up and down the parse tree. The default access modifier for BNF productions is public.

There are two parts on the right hand side of a BNF production. The first part is a set of arbitrary Java declarations and code (the Java block). This code is generated at the beginning of the method generated for the Java non-terminal. Hence, every time this non-terminal is used in the parsing process, these declarations and code are executed. The declarations in this part are visible to all Java code in actions in the BNF expansions.

XUL

XUL (pronounced zool) was created to make development of the Mozilla browser easier and faster. It is an XML language so all features available to XML is also available to XUL.
Most applications need to be developed using features of a specific platform making building cross-platform software time-consuming and costly. A number of cross-platform solutions have been developed in the past. Java, for example, has portability as a main selling point. XUL is one such language designed specifically for building portable user interfaces. It takes a long time to build an application even for only one platform. The time required for compiling and debug can be lengthy. With XUL, an interface can be implemented and modified quickly and easily.
XUL has all the advantages of other XML languages. For example XHTML or other XML languages such as MathML or SVG can be inserted within it. Also, text displayed with XUL is easily localizable, which means that it can be translated into other languages with little effort.

XUL provides the ability to create most elements found in modern graphical interfaces. Some elements that can be created are:

- Input controls such as textboxes and checkboxes
- Toolbars with buttons or other content
- Menus on a menu bar or pop up menus
- Tabbed dialogs
- Trees for hierarchical or tabular information
- Keyboard shortcuts

IX. ADVANTAGES:

1. Reduces programming effort
2. Increased maintainability
3. Easy error detection and correction
4. Multi target

CONCLUSION

The system produced satisfies the requirements outlined at an early stage in the project.The system is able to successfully convert MAGIC Script into target language specified. The target languages used are Java AWT, Java Swing, and XUL and C #.Net.MAGIC Script is able to fulfill its role of an intermediary between basic programming constructs and high level programming constructs which most of the user are unaware off. It also provides support to a relatively newer language XUL whose constructs are still unknown to most of the user.The IDE provides required help in form of auto generated MAGIC Script on a single click of mouse.The compiler generated is truly scalable and can incorporate newer languages just by adding some new files.

REFERENCES

[1.] Andrew Appel, Jens Palsberg: Modern Compiler Implementation in Java. Cambridge University Press, 2nd edition, 2003.

[2.] Keith Cooper, Linda Torczon: Engineering A Compiler. Morgan Kaufmann, 2004.

[3.] Niklaus Wirth: Compiler Construction. Addison-Wesley, 1996.

[4.] Michael Morrison: Java, Second Edition

[5.] Bruce Eckel: Thinking in JAVA

[6.] Adrian Kingsley-Hughes: C# 2005 Programmerøs Reference. Wiley Publishing Inc.
[6] Amruta Mukund Talathi, 2013 A Java Based 4th Generation Multi-Targeted User Interface Compiler (JUICE) International Journal of Innovations in Engineering and Technology (IJIET)

[7.] Java.net: JavaCC [tm]: Grammar Files

[8.] XulPlanet.com: XUL Tutorial

ACKNOWLEDGEMENT

Pankaj P. Babhare and Khushbu W.Patre presenting International Paper in December 19,2013 at KITS, Ramtek on topic Magic: Multitarget Graphical user Interface Compiler.

Pankaj P. Babhare presenting International Paper in January 13,2014 at P.R. Pote Patil College of engineering Amrawati s on topic Magic: Multitarget Graphical user Interface Compiler.

Pankaj Babhare and Khushbu Patre presented so many national level paper presentation.