Wrapping C++ and Java Using a Meta framework

Nadera Beevi
MCA Department
TKM College of Engineering
Kollam, Kerala, India
snnadera06@gmail.com

Chitra Prasad
Department of Computer Science and Engineering
TKM College of Engineering
Kollam, Kerala, India
dcpvenus@yahoo.com

Vinod Chandra
Computer Centre
University of Kerala
Trivandrum, India
vinodchandrass@gmail.com

***Abstract* – This paper describes a parser developed for a Meta framework obtained by combining C++ and Java language segments. It enhances flexibility and effectiveness of Execution Preserving Language Transformation (EPLT) using Meta framework. Augmented versions of existing languages can be developed by combining good properties of two languages. The growing popularity of Java language forces programmer to implement data structures and algorithms of other languages in Java. The framework identifies and parses source code with C++ and Java language statements using metagrammar developed and create a unified AST for the hybrid source code. Bytecodes are generated for the AST and interpreted. It provides a transformational scheme where many error prone tasks are automated so that a more powerful robust incremental compiler can be developed.**

Keywords - Compilers; language constructs and features; parsing; lex; yacc

I. INTRODUCTION

Compilers are used to translate a source program into target program. It can also be considered as a program that defines object language in terms of source language. Multiple keyword constructs used in a single program is came to be known as a Meta language. It is a program transformation methodology for developing efficient programs by combining good properties of different languages. *Meta* is a large research project whose main aim is to increment and unify the syntax and semantics of existing languages [1]. It acts as a language template and develops a new language when Meta syntax is applied to existing object oriented languages. We have developed a framework from Java and C++ which is designed in such a way that class developed in one language can be used in other language and vice versa. It provides better flexibility for programmers in using various language constructs.

The use of programming languages like Java which provides type safety, pattern matching and automatic storage management can reduce the code size of compiler and eliminate some common kind of errors. The portability of Java programs and its support for Internet applications made Java one of the most useful programming languages in the world. C++ is a hybrid language with object oriented features added to the procedural concept. The presence of pointers in C++ allows the user to access memory locations directly and the lack of garbage collector requiring explicit deletion of memory areas make the language to be termed as unsafe. But Java offers better solution to this problem. Since a lot of software is written in C++, it would be really helpful if the unsafe code is converted to a safe equivalent. One method is to invoke unsafe code from a safe environment is by integrating native methods in language like C with Java using Java Native Interface (JNI) and Cygnus Native Interface (CNI) [2]. JNI provides interface that enable communication between these two languages. Even though performance gain is achieved by integrating programs compiled to native machine language with Java, security and portability threat persists. Security threat leads to buffer overflow and heap corruption attacks. Portability threat is due to the fact that since C code is compiled to native machine language of a particular platform, platform independence of Java is lost.

Our goal is to develop a Meta framework for a program containing keyword constructs from C++ and Java. The input to our system is a precise description of source and target language. The syntactic and semantic information about the component languages of Meta are mixed to make the Meta parser easy to understand and maintain. The syntactic elements of the language are considered as strings and grouped into syntactic structures. Context free grammars are used to describe the structure of programming languages. Syntax and Semantics of two

languages are clearly defined by Recursive Functions on Context Free (CFRF) languages through which syntax check of the language can be done [3], [4]. Semantic analysis of the language can be done through attribute grammars. The system uses syntax equations resembling BNF into which output intermediate language commands are inserted. Each syntax equations consists of embedded code generators which test the input string for a particular syntactic structure and delete if found. The following section gives a brief review of the related study.

II. RELATED WORKS

The purpose of Execution Preserving Language Transformation (EPLT) is to translate a program such that the target program executes identically as the source program. It is a standard technology for software maintenance and evolution [5]. Source to source program transformation involves complex transformations that results in the modification of data/control flow to produce semantically and functionally equivalent output.

A number of related works performing source to source program transformation is discussed here. *MoHCA*-Java is a tool which translates a pure C++ file to Java code [6]. Abstract syntax trees of C++ code is generated which is analyzed using Gen++ program. It searches for target patterns in the tree using a rule based language. If patterns are found changes are outputted as text messages prompting the user to implement the change. So, active involvement of the user is required before compiling to Java. In *Cappuccino* also, a pure C++ file is translated to Java equivalent [7]. In this work, keywords are handled by replacement lists which contain information for replacing keywords found in the source file. A partial translation was carried out in their work. Messages are inserted in a separate file to mark those parts that must be edited manually. Similar translators are designed in C2j and C2j++ which all rely on text stream processing [8]. C2j performs partial parsing on C++ files generating Java source code. Language C2j++ access C++ classes using Java native method interface thus affecting security and portability of Java programs.

Many developers consider source to source program transformation as expensive, time consuming and therefore infeasible. So one solution proposed is to wrap it and embed it in new application without changing the language rather than redeveloping [9], [10]. Depending on the application, interface has to be designed. It is quite cheaper and provides a faster transition.

There have been a few attempts of implementing language transformation using meta frameworks. Constructs are added in ALGOL to include new data types and operators and to redefine the behavior of existing operators [11]. The new features are included as replacement rules to reduce storage overheads.

MetaL developed for C and Pascal implements a meta framework for two procedure oriented languages [12]. Syntax checking of the source code is done using meta grammar developed from C and Pascal grammars. The meta program is tokenized, parsed and executed using *lex* and *yacc*. The basic intention is to provide flexibility in using data structures for programmers. Any change requiring a design decision is not recognized or reported.

Reeuwijk talks about a template based meta compiler for generating source code of any programming language [13]. In this work a template language **Tm** was developed. It accepts data structure definitions and source code template as input and produce target code of a particular programming language as output. It supports compile time meta programming where data structure definitions are provided at compile time itself. File inclusion is supported so code can be shared between programming languages.

We have developed an automated approach for interpreting hybrid source code containing C++ and Java language segments and the next section describes the structure of the Compilation Model developed focusing on the Meta grammar of various data and control structures of two languages.

III. IMPLEMENTATION

Program transformation frameworks promote pure code migrations and help to reengineer legacy systems to object oriented platforms. The input of the system is a meta program containing C++ and Java language constructs. Source program expressed as sequence of characters is translated to a representation for use in the meta framework. It is a precise description of source and target languages. The source language is specified using a context free grammar, meta grammar and a *lexing* and a *parsing* method can be used to perform the translation.

Figure 1 shows the Compilation model developed. The source program can be written in C++, Java, or combined statements.

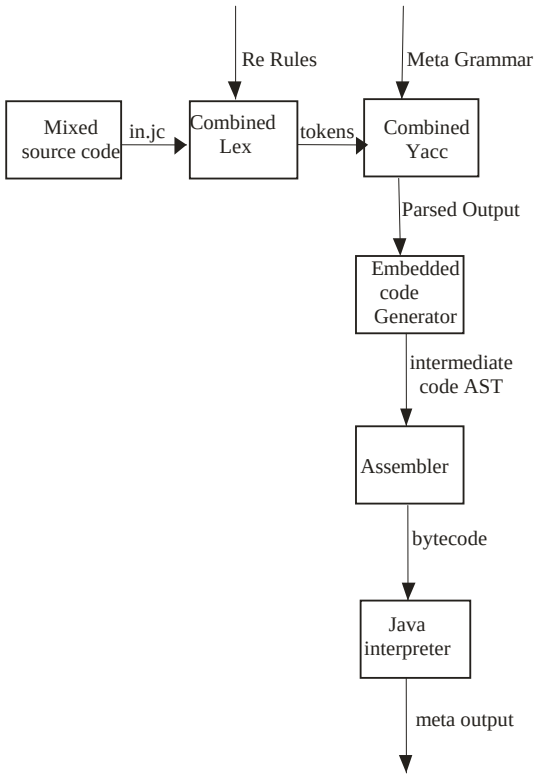

Figure 1. Meta Platform Structure

The lexical patterns in the meta program are identified using *lex* tool. The rule section of lex file defines meta *lex* rules for C++ and Java tokens. Lexical analysis is performed using these rules and tokenizes the source code. If the string in source code doesn't match with the rules, error routine is called by *lex* pre-processor.

A *lex* program to identify data types and variables in the source program, calculates line count and returns pointer character present, to the parser as follows.

```
UCN  (\\u[0-9a-fA-F]{4}|\\u[0-9a-
fA-F]{8})
   %{
        int lineCount=0;
   %}

%%
( _a-zA-Z]|{UCN})([ _a-zA-Z0-9]|
{UCN})* { printf("\n IDEN" };
int|short|long|char|boolean|float|
double { printf ("TYPE" };
[\n]  { lineCount++ };
[*&]  { return *yytext };

%%
```

A preprocessing of input file is done by lex preprocessor before passing input to the parser. Some preprocessing includes enclosing main function in C++ in a class file and the translations of definitions into constant variables. The processing of enumerated data types are also done during preprocessing stage. For example

```
#define PI 3.14
```

changed as

```
public static final float PI=3.14;
```

and

```
typedef int a[3][3];
 a b;
```

changed as

```
int b[3][3];
```

The preprocessed output is used by *Yacc* to generate tokens from lex file. *Yacc* is a tool that generates syntax analyzer for the source code. The tokens returned by lexical analyzer are analyzed by *Yacc* using a set of meta grammar rules written in BNF notation and generates a parse tree. Each time a variable is found, parser performs a semantic action which adds the identifier, its value and scope to the symbol table. At the end of parsing these inputs are ready for the remaining phases of compilation. We use *gcc* system for the remaining phases.

A meta-grammar is a combination of C++ and Java grammar. The correctness of compiler purely depends on a set of grammar rules that are written in language of formal mathematics. Strings which are sentences of the target language are accepted by *Yacc* and report an error when a syntactically incorrect symbol is encountered. Various types of conflicts arise when we try to combine grammars of different source languages. Around 100 shift reduce or reduce-reduce conflicts identified while parsing the input. Slight changes in production rule disambiguate the grammar and parse input correctly. For example if we try to include global variables and methods in the source code, shift reduce conflict is identified since the grammar supports variables and methods declared in classes and by including EXTERN keyword along with the production, it can be removed.

```
type_declaration:
class_declaration
|interface_declaration|EXTERN
local_variable_declaration_statement
|EXTERN method
```

The definition section of *Yacc* file is concerned with token definitions. The rule section provides grammar rules resembling BNF with code defining clauses added. Actions can be written at the end of grammar rule or in middle. Non terminals on left hand side of the rule are matched with return codes from *lex*-tokenizer. They form % token definitions in the *Yacc* file. The matched tokens are pushed to the stack and reduced with corresponding non terminal of the rule. They act as recursive recognizers

with embedded code generators when executed. It refers each token identified in the right hand side as $1,$2,$3.. and $$ is used to set the value of left non terminal. Initially $n quantities are pushed to the stack and later removed when they can be replaced by $$ which becomes the top of stack. The process repeated when the final start symbol of grammar reaches the top of the stack. The grammar for evaluating expressions is shown below.

For example

```
expr: expr'+' mulex {$$=$1+$3;}
     | expr'_'mulex {$$=$1+$3;}
     | mulex { $$=$};
```

Meta parser provides data structure flexibility to the user. It combines several features found in C++ and Java into one language. It resolves syntactic and semantic differences between two languages using meta grammar and cleverly handles design issues like multiple inheritance, pointer usage, virtual functions etc. which exist in C++ but not in Java. C++ provides flexible I/O mechanisms for the user. The compilation model we developed maps them and makes it compatible with Java system so that undetected error conditions causing security problems can be reduced. The framework supports automatic garbage collection so that explicit deletion of variables using delete operator is not needed as in C++. Unused memory is returned to the operating system through a garbage collector so that errors caused by memory leaks and dangling pointers are prevented. The use of meta grammar in handling various design issues between the languages are discussed.

A. Pointers

Pointers are deliberately avoided in Java to prevent user from accessing the memory directly which creates dangling pointers and causes memory leak. But it allows the programmer to write code for flexible memory operations. It can be used to access memory address from outside a program's code and data. The unauthorized access can be prevented by restricting the use of pointers within the limits of java run time system. The meta grammar for incorporating pointers in the source code is as follows:

```
type : primitive_type
|reference_type
reference_type: cp_pointer_type
| cp_struct_type
cp_pointer_type :  primitive_type *
variable_decl  :
type variable_declarators   SEMICOLON
variable_declarators  :variable_decl|
variable_declarators COMMA
variable_declarators.
```

While creating Abstract Syntax Trees (AST), a pointer table is used to store the identifiers declared as pointers. When an address assignment occurs to the variable, corresponding index of the symbol table is stored in the pointer table. Using this index all subsequent pointer arithmetic is implemented. The same concept can be used to pass parameters to functions using pointers.
Another efficient use of pointers is in traversing array sequentially. It provides significant performance boost rather than using array index. The following code using pointers for traversing array can be translated to our framework as shown.

```
int *k;
int arr_int[100];
k=arr_int;
k++; *k=10;
```

```
int []k,_idx_k=0;
int []arr_int=new int[100];
k=arr_int;
_idx_k++;
*k => k[_idx_k]
*(k+10) => k[_idx_k+10];
k++ => _idx_k++;
(*k)++ => k[_idx_k]++;
```

B. Virtual Functions

Runtime polymorphism in C++ is implemented through virtual functions declared in base class and overridden in subclasses. The information about the receiver classes of dynamically dispatched messages is obtained by analyzing the inheritance structure of the source code. In our system while creating Abstract Syntax Trees (AST) the derived objects are statically determined by the system and virtual calls are replaced with direct procedure calls. If the method is overridden in a single subclass, the virtual call is statically linked, reverting virtual function to a non virtual one. Otherwise a series of runtime class tests is done for the expected receiver classes and replacing the virtual method call with a direct procedure call to the corresponding class found. The following algorithm identifies the receiver classes of dynamically dispatched methods.

```
for each class C in the program
  { for each method m in the class
   { insert m () to the method
        list of the class
     if it is virtual set vflag for
        the method
     if it is in the method list of any
     other class P,
     Add P to the class list of
        method m }    }
```

Class hierarchy analysis helps to identify methods visible to a particular class. Union of method sets is done to find methods visible in derived classes. The intersection of the method set is performed to identify the class set where the method is visible. A base pointer table is created for each base class used in the program. It contain information about the derived classes of particular base class. The following meta grammar identifies virtual method call and generates a dynamic replacement strategy that invokes method of the corresponding receiver class.

```
Cpp_point_type : MULT name
type_decl: type cpp_point_type

 { if type ∈  base_class

   base_obj=name;}
```

Since the information about the derived object is not available at the usage site dynamic modification of the base pointer is needed.

C. Structures and Classes

To include *structure* and *class* types in the meta program the following metagrammar is used.

```
Compilation_unit :
package_declaration_opt
|cp_struct-declaration_opt
|interface_declaration
|jc_class_declaration; cp_struct-
declaration_opt : cp_struct-
declarations;
cp_struct-declarations :
cp_struct-declaration|
cp_struct-declarations
cp_struct-declaration : STRUCT
IDENTIFIER LBRACE cp_struct_def
RBRACE cp_struct_def :
|cp_struct_def  type
variable_declarators SEMICOLON.
```

D. Data types

The major problem in dealing with data types of both languages is some data types of one language have no direct equivalent in other. For example, Java doesn't support unsigned data types, but C++ supports. Float data types in C++ has no direct equivalent in Java. The unsigned data types in the source code are implemented using the next higher signed data types of Java.

For example

Short n= (short) (a&0xff) where a is an unsigned byte data type of C++.

E. Exception Handling

We can write meta grammar rules for all data structures and control structures. Exception handling mechanism is implemented in Java and C++ to reduce run time errors. But user attracts the way it is implemented in Java. Both language implementations can be used in the source code using the meta grammar as

```
statement:throw_statement|try_statement
throw_statement:THROW expression
                SEMICOLON
method_def : modifiers method_decl
             throws_opt
throws_opt : |throws
throws : THROWS class_list
```

F. Break and Continue

Nested loop handling with labeled break and continue statements can be used along with equivalent C++ constructs with slight change in the production rule. The corresponding meta grammar is

```
statement: break_statement;
break_statement: BREAK
    identifier_opt SEMICOLON
identifier_opt: |IDENTIFIER;
```

which support unlabeled and labeled breaks.

The meta grammar created in Backus-Naur Form (BNF) like rules which incorporate embedded code generators that analayze the source program to identify classes, variables and methods. Usually structures are transformed into classes with data members qualified as public. Most of the legacy applications use union to implement alternate views of memory. Each member of the union form subclass of a super class with other members of the structure other than union members. A member of the super class is used as a flag to identify which member of the union to be used by instantiating the corresponding subclass [14]. Separate variables are used to keep track of whether the control is inside the main or inside a particular class. Global variables in the input code are eliminated in the object program since it affects the data hiding properties of object oriented system. They are converted into instance members of

the class with public access specifiers to make it visible in the current package. The class also encapsulates functions referring these variables. The transformation procedure to convert structure with pointer type to class as follows.

```
 struct student_rec
   {
       char  * name;
       int  roll;
       int  marks[3];
       struct student_rec *next;
   }
class student_rec
       {
          String name;
          int roll;
          int marks = new int[3];
          student_ rec  next;
       }
new_rec= new student_rec;
```

The embedded code generators in the metagrammar generate abstract syntax tree of the meta program. Bytecodes are generated from AST and interpreted. Data flow optimizations such as constant propagation, constant folding, algebraic simplification can improve execution time performance of bytecodes. Constant propagation identifies constant value of a variable and propagate them to the place where variable occurs and replace it with corresponding value. It helps in simplification of algebraic expressions by evaluating expressions at compile time. It greatly reduces execution time evaluations especially inside a loop. Constant propagation optimizes local loads by generating bytecodes with native operands. Data flow optimizations at source level is applied through metagrammar while generating Abstract Syntax Trees. Method inlining and eliminating unnecessary exception checks also optimizes the performance of bytecodes.

Our aim is to develop a new language which targets the virtual machine. A Java .class file is not easy to generate. So AST is processed to produce ASCII descriptions of Java classes which form the input of a Java assembler. It converts them into binary Java class files which can be executed by the Java Run Time System. The Meta platform provides a simple way to identify source actions that are relevant to a particular grammar rule. The user is flexible enough to use the features available in both languages to implement a particular behavior. That makes the system accessible and feasible for people for whom many constructs of two languages are out of reach so that learning curves are reduced. The analysis and evaluation of program specification is simplified through lex and yacc. It supports code starting with C++ or Java.

IV CONCLUSION

We have described the concepts behind the Meta parser developed using meta grammar. It parses almost all statements in Java and C++ and assist in the creation of bytecodes. Although many of the details of the theory are beyond the scope of this paper, the Meta platform developed is valuable even apart from the theory for the following reasons. (1) It provides data structure flexibility for programmers. (2) The unification of two or more existing languages makes the design and implementation of applications easier. (3) Augmentation of existing languages helps to identify synergism between various language features. (4) Since the features of existing languages are combined, learning curves are reduced. The implementation of program transformation through *lex and* yacc translate unsafe source code containing C++ and Java constructs to safe byte code using the metagrammar developed. Reengineering of legacy systems are needed to make it internet compatible. So the target of the developed meta language compiler is a virtual machine. The data and control flow of the meta language is modified to obtain functionally equivalent output.

REFERENCES

[1] W. Holst, "Meta: Extending and Unifying languages," OOPSALA'04, 2004, pp. 331-344.

[2] Sheng Liang, The Java Native Interface. Programmers Guide and Specification, AddisonWesley, Longmam,Inc, 1999.

[3] Y. Dong, " Recursive functions of Context free Languages(I) – The definitions of CFPRF and CFRG," Science in China, Series F, 45(1), 2002, pp.25-39.

[4] Y. Dong, "Recursive functions of Context free Languages(II) – Validity of CFPRF and CFRG definitions," Science in China, Series F, 45(2), 2002, pp.1-21.

[5] R.M. Balzer, N.M. Goldman, and D. S. Wile, " On the Transformational Implementation Approach to Programming," In Proceedings of the 2nd International Conference on Software Engineering, 1986, pp.337-344.

[6] S. Malabarba, P.T. Devanbhu and A. Stearns, " MoHca Java: A tool for C++ to Java Conversion support," In Proceedings of International Conference on Software Engineering, ICSE,1999.

[7] F.Buddrus and J. Schode," Cappuccino - a C++ to java translator," In Proceedings of the 1998 ACM Symposium on Applied Computing,1998.

[8] G. Laffra," C2J: a C++ to Java translator," Novosoft, 2001 .

[9] H. M. Sneed, " Wrapping legacy COBOL programs behind an XML interface," In Proceedings of WCRE , IEEE Computer Society Press, Stuttgartt, 2001.

[10] H. M. Sneed, "Migrating from COBOL to Java," In Proceedings of International Conference on Software Maintenance (ICSM), IEEE, Timisoara, 2010, pp.1-7.

[11] B. Galler and A.J. Perlis, "A Proposal for definition in ALGOL," ACM Communication,1967, pp.204-219.

[12] S.S. Vinod Chandra and Achuthsankar, S. Nair, " A MetaL for C and Pascal,"SIGCSE Bulletin,ACM,2007, 39, 4.

[13] C. Van Reeuwijk," Rapid and Robust Compiler Construction Using Template-Based Metacompilation," Springer Verlag Berlin Heidelberg,2003, pp.247-261.

[14] Mariano Ceccato, Thomas Roy Dean, Paolo Tonella, And Davide Marchignoli, " Data model reverse engineering in migrating a legacy system to Java," In Proceedings of 15thWCRE, 2008.

Authors' information

[1] **Nadera Beevi** (corresponding author)
Assistant Professor
Department of Computer Applications
TKM College of Engineering
Kollam, Kerala
India.

[2] **Dr. Chitra Prasad**
Professor and Head
Department of Computer Science and Engineering
TKM college of Engineering
Kollam, Kerala
India.

[3] **Dr.Vinod Chandra**
Director
University of Kerala
Trivandrum, Kerala
India.

Nadera Beevi Assistant Professor, Department of Computer Applications (MCA), TKM College of Engineering, is pursuing her Ph.D. from Kerala University. She received her B.Tech in Computer Science & Engineering from TKM College Of Engineering and her ME from PSG College of Technology, Coimbatore. She is a member of ACM. Her research areas of interest are compilers and optimization in compilers. Email: snnadera06@gmail.com

Dr.Chitra Prasad , Professor, Department of Computer Science and Engineering is associated with TKM College of Engineering since 1983. He received his B.Sc. (Engg.) in Electrical Engg. from University of Kerala, M.Tech. in Computer Technology from IIT Delhi and Ph.D. in Computer Science and Engineering from IIT Kharagpur. He was Chairman, Board of Studies (Engineering) University of Kerala from 2007-2010. He is the Chairman, Board of Examiners (B.Tech Degree Exam in Computer Science) of Kerala University from 2004 onwards. He has published many papers in international journals and national and international symposiums. His areas of interest include compilers, computational geometry and design and analysis of algorithms. He is a member of IEEE, Computer Society of India, Indian Society for Technical Education, and Institution of Engineers. Email: dcpvenus@yahoo.com

Dr. Vinod Chandra working as the Director, Computer Centre, University of Kerala. He had his B.Tech in Computer Science & Engineering from University of Calicut and M.Tech in Software Engineering from Cochin University of Science and Technology, Kerala. He had his Ph. D. in Engineering & Technology from Department of Computational Biology, University of Kerala. He holds an M.B.A degree from IGNOU, New Delhi in Operations Management. Since 1999, he has taught in various Engineering Colleges in Kerala State. He authored one book and a modest number of research publications with high impact factor in International level. He is a reviewer of many international journals and conferences. He is an associate editor of International Journal of Computational Intelligence Techniques. His research area includes Compilers,Machine learning algorithms and Computational Biology. He works in the M.I.T, Government of India funded research projects. He holds many consultancy activities include CDAC, Thiruvananthapuram, CSSB, Thiruvananthapuram. His collaborated institutions include Rajiv Gandhi Centre for Biotechnology, Thiruvananthapuram, and various companies in Technopark, Thiruvananthapuram. He is a member of IEEE, Computer Society of India, and Indian Society for Technical Education, and Institution of Engineers. Email: vinodchandrass@gmail.com