IOSR Journal of Computer Engineering (IOSR-JCE)
e-ISSN: 2278-0661,p-ISSN: 2278-8727,
PP 21-25
www.iosrjournals.org

Development of Intermediate Model for Source to Source Conversion

Ms. Naziya Shaikh[1], Prof. Manisha Naik Gaonkar[2]
[1, 2] *(Computer Department, Goa college of Engineering, Goa university, India)*

ABSTRACT: *This paper proposes the idea of conversion of source code from one programming language to another while maintaining the same correct functionality using a new form of intermediate language approach. The processing of such an intermediate language file can provide efficient conversion between two programming languages and can also be extensible to multiple languages. The proposed idea of a common intermediate file template is expected to perform better than other forms of intermediate language that have been used for conversion like xml file, style sheets or other methods like direct conversion methods.*
Keywords - *High Level Languages; Intermediate Approach; Java; PHP; Programming Languages; Source Code Conversion; Source to Source Translation*

I. INTRODUCTION

A large number of programming languages have been developed with more advanced features being included into each new language. Each language has its own set of defined keywords, and valid syntaxes and grammars. Conversion of source code from one programming language to another has become inevitable in certain situations for industrial as well as educational purposes. Consider for example, a company wants to use the opportunities and features provided by the new languages that were not available in the earlier languages that the company has already used to develop a certain product. Company will have to again invest into workforce for sole purpose of translating from one language code to another for upgrading the given product. Also it is a very time consuming process. Such a process would require large amount of resources, time and money. With the availability of translation tool this overhead can be reduced to a great extent. Such a tool would be a better option for companies as development of a compiler for conversion between languages would be quite expensive and would require high amount of processing. Such a conversion could also help in reusability of the code sections written in any language and also prevent errors that occur during manual translations. Lot of research has been done on such source to source translators.

II. RELATED WORK

Various techniques for conversion of source code from one programming language to another have been developed. Not much success has been achieved as yet with the target code produced by the systems. Most of the systems need post editing to access the language specific features of target language and also target language code lack structure and readability. Initially, research was based on direct approach where a large code was written for dynamic conversion using grammars of the source and the target programming language. The paper [1] uses an abstract idea about such a concept of language grammars. It uses annotated grammar to produce partial automated translations between operations languages which are used for spacecraft language procedures. In [2] translation of various programming languages source code to ADA language has been considered. The paper agrees upon certain limitations to such system that some codes in other languages may be difficult to translate into ADA and also mentions that the code requires post editing to take advantage of various features of ADA.

Another approach used when it is difficult to handle the case where certain codes in the source language do not have corresponding features available in the target language is the use of subset of the given language. We first reduce the given code into a subset which has translation available in the target language. Such an idea is used in [3] where translation takes place between ADA and Pascal. For translation, subsets are created and the translation is done based on these subsets of the languages. The disadvantage of such method as mentioned in [4] is that sublanguage definition is not an easy task and the algorithms work well only for simple codes. As the complexity of the code increases, the structure of the target program is affected.

Later there were advances in research over intermediate method for conversion. As mentioned in [5], an approach based on abstraction and reimplementation can be adopted for the program conversion. Source program is first examined to get an abstract model based on the overall functional understanding the code. This model needs to be language independent. From this model, further conversions are done to the target code while

keeping the basic information intact and discarding the extra information not needed for conversion. Use of intermediate language for conversion has been discussed in [6] where the authors have suggested a recursive approach which also involves optimization of the code. Various forms of intermediate methods evolved like using xml as an intermediate language, using databases to store intermediate information, using style sheets, etc. There are various issues in the design of an intermediate code as mentioned in [4] mainly because of the differences in the language features which may not be supported in the target languages. Intermediate code should be formed such that functionality of the source language features that are not present in the target language should be simulated using combination of target language features or libraries.

In [7], the idea of using xml as an intermediate language has been studied and tested using the case study of conversion between java and C++. It was found that while the use of xml makes the intermediate language easy to understand and code for, it introduces various parsing delays, as large xml files are created for even a small program with minimal lines of code. Xml style sheets could also be used as a form of intermediate language. Such an idea is presented in [8] where a new approach of mapping style sheets is used. Each style sheet corresponds to one specific language and using the mapping of the style sheets, a generic translator translates from one programming language to another based on the corresponding style sheets. If we use databases as an intermediate storage, then retrieving the records while generating the target code would require very high level of processing. If we use a defined template in the form of a file as an intermediate code, and use that file to create an intermediate file for each input program, then, the standardized template could easily be used by the target language generator programs and also the execution of the translation would speed up as there is no parsing required like in xml file and also there is not much processing required while accessing the data in the file line by line based on template.

This paper uses the idea of new template for an intermediate file similar to the UNL notation which saves all the relationships of a statement in natural language into a file with a given predefined set of attributes to describe the meaning of the sentence as mentioned in [9][10][11]. Similarly a predefined template could be used to convert one programming language to another with ease and could be used to build a language converter tool with lesser processing overhead. This template would also facilitate extension of the same conversion process for other languages as target. The template is formed in such a way that the language specific features of the target language can be easily derived based on the intermediate file and the target language grammar.

III. PROPOSED WORK

The overall design of the proposed translator system is as shown in the figure below. Fig. 1 uses two specific languages java as source programming language and PHP as a target programming language as an example, but the system is extensible to multiple languages.

The system consists of 2 modules. The first module is for the generation of intermediate file from the source code file. The source language will be parsed based on the grammar for source language and corresponding intermediate code for that grammar will be generated. The second module is the mapper program that contains the codes to map the generated intermediate file with the target language and generate the target language source code.

The intermediate language considered is in the form of a file. An initial template is determined for every construct in the programming languages. Based on this template, module 1 will generate an output in the form of an intermediate file, which will follow the initial template specified. This initial template remains common and the same template can be used while creating generator programs to generate source code from the intermediate code.

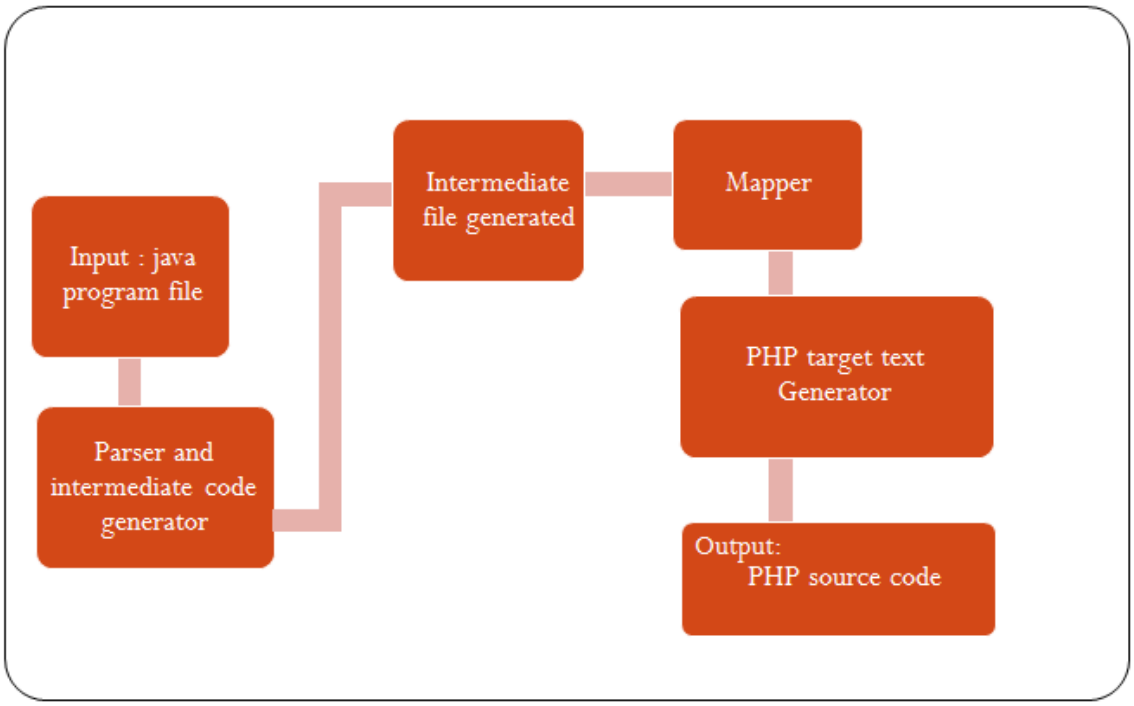

Fig.1. Basic Design of Language Convertor

Consider the following example program.

```
public class A
{
        int a, b;
        public void function1(int g)
        {
                System.out.println("Hello "+g);
        }
}
```

We will initially define a grammar such that it will accept inputs as valid java constructs. For example, for the above program, we can have grammar rules as:

program -> import_statement | class_statement
class_statement -> [access_modifier]? class_key class_name classbody
access_modifier->'public' | 'private' | 'protected' class_key -> 'class'
class_name -> identifier
identifier -> letter (digit | letter)* ('_')*
classbody -> { mainbody }
mainbody -> data_part function_declaration | function_declaration data_part
datapart -> [access_modifier]? Datamember
datamember -> data_type datavar
datavar -> identifier semi_colon | identifier coma datamember
semi_colon -> ';'
coma -> ','
function_declaration ->access_modifier return_type identifier'(' [parameterlist]? ')'

Further there will be grammar to specify parameter list and so on. We start by using a parser (lex and yacc) and detect the grammar patterns for each line using similar grammar rules as specified above. Once we obtain the patterns a code will be written in the action part of the parser pattern match section (rules section) so that we can create a unique intermediate language. For example, for the program given above, based on pattern matching, we will create the following unique intermediate language:

```
Class(A,public,1,2,1)
OpenBracket(class,A,1)
DataMember ( int ,a, , class, class, A, no, no,0 )
DataMember ( int ,b, , class, class, A, no, no,0)
FunctionMember(public, void, no, no, function1, 1, 6)
DataMember ( int ,g, , function, function, function1, no, no ,0)
OpenBracket(function,function1,5)
Statement(print, 2,, [9,6], no, function, function1, 5)
DataMember ( string ,temporary_var, , function, function, function1, no, no, hello )
CloseBracket(class,A,1)
```

Once we obtain this intermediate language conversion for our source program, we can then use the grammar rules that we will specifically define for the target language to generate the target text. Using this method, we can therefore access the language specific properties of the target language while generating the target code. For example, for PHP language we can have the following grammar to match against the lines in intermediate language:

```
program -> import_statement | class_statement
class_statement -> [access_modifier]? class_key class_name classbody
access_modifier->'public' | 'private' | 'protected'
class_key -> 'class'
class_name -> identifier
identifier -> letter (digit | letter)* ('_')*
classbody -> {  mainbody }
mainbody -> data_part function_declaration | function_declaration data_part
datapart -> [access_modifier]? 'var $' datamember
datamember -> identifier (semi_colon | coma datamember )
semi_colon -> ';'
coma -> ','
function_declaration ->'function' identifier '(' [parameterlist]? ')'
```

The following code will be generated finally for the PHP program.

```
<?php
public class A
{
        var $a, $b;
        function function1($g)
        {
                echo("Hello "+$g);
        }
}
?>
```

We can generate target language source codes in similar way for java programs with different types of constructs like if, else, etc. We can also extend the system to obtain source code in other languages if for example we use grammar of the other language instead of PHP and develop a code for the mapper in that language. We can also extend the mappers module to create a unique generator class that would take as an input the target language and then generate the mapper program for the given language based on the grammar for the language and the intermediate file template. With this the development effort of the system could be highly reduced.

IV. CONCLUSION AND FUTURE WORK

Conversion of high level source code from one language to another is indeed a very challenging task. Various solutions have been suggested to achieve success and accuracy in this task. This paper proposes the idea of a defined intermediate template file for extensible translation between multiple programming languages. This template file would be used by first module of translator system to generate intermediate file for each input program based on the grammar of the source language. The intermediate representation is easy to understand, making it easier for development of the system and coding. Based on the intermediate code, the generator would

generate the target code for the target programming language. Use of such a file would reduce processing effort since lesser amount of parsing is required. Also such a model of the system is extensible as it uses the idea of abstraction to a common form and can be used to convert into any form required. Also for those language constructs whose corresponding constructs are not present in the target language, we can define intermediate language templates in such a way that we are able to simulate the required conversions in the target language. The future work will involve development of the different languages and creation of a generator program that would by itself create the mapper programs if we give the target language name based on the grammar available.

REFERENCES

[1] Diego Ordonez Camacho, Kim Mens, APPAREIL: A tool for building Automated Program Translators Using Annotated Grammars, Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008), pp. 489-490, 2008.
[2] P. J. L. Wallis, Automatic Language Conversion and Its Place in the Transition to Ada, Proc. 1985 annual ACM SIG Ada international conference on Ada (SIGAda '85), Vol. 5, No. 2, 1985, pp. 275-284.
[3] P. F. Albrecht, P. E. Garrison, S. L. Graham, R. H. Hyerle, P. Ip, B. Krieg-Brückner, Source-to-Source Trans- lation: Ada to Pascal and Pascal to Ada, Proc. ACM-SIGPLAN symposium on Ada Programming language (SIGPLAN '80), Vol. 15, No. 11, 1980, pp. 183- 193.
[4] David A. Plaisted, Source-to-source translation and software engineering, Journal of Software Engineering and Applications, 2013, 6, 30-40.
[5] Richard C. Waters, Program Translation via Abstraction and Reimplementation, IEEE transactions on Software Engineering, 1986.
[6] Dony George, Priyanka Girase, Mahesh Gupta, Prachi Gupta, Aakanksha Sharma, Programming Language Inter Conversion,2010 International Journal of Computer Applications (0975 - 8887)Volume 1 – No. 20.
[7] Mohammed Salih, Ognyan Tonchev, High Level Programming Languages Translator, Master Thesis no.MCS-2008-17,Blekinge Institute of Technology, Sweden, 2008.
[8] David P. Clark, Min Chen, John V. Tucker, Automatic Program Translation—A Third Way, Proc Proc. IEEE Sixth International Symposium on Multimedia Software Engineering (ISMSE'04), 2004.
[9] Sameh AlAnsary, Interlingua-based Machine Translation Systems: UNL versus Other Interlinguas, Proc. 10th Egyptian Society of Language Engineering Conference, Ain Shams University, Cairo, Egypt, December 15 – 16, 2010.
[10] Pushpak Bhattacharyya, Multilingual Information Processing Through Universal Networking Language, Indo UK Workshop on Language Engineering for South Asian Languages, 2001.
[11] Hiroshi Uchida, Meiying Zhu, "The Universal Networking Language Beyond Machine Translation", proc. International Symposium on Language in Cyberspace ,UNDL Foundation September 26, 2001.