

Source Code Analyzer and Translator

Rajeev Bedi¹, Vinay Chopra², Dinesh³

¹CSE Department, BCET Gurdaspur,

²CSE Department, ³IT Department, DAVIET Jalandhar,

rajeevbedi@rediffmail.com, vinaychopra222@yahoo.co.in, erdineshk@gmail.com

Abstract- *Source code analysis is technology aimed at locating and describing different tokens, classes, libraries, packages, inheritance level etc and areas of weakness in source code. Those weaknesses might be security vulnerabilities, logic errors, or many other types of problem-causing code.*

At time of writing, 80% of the Fortune 500 have already deployed, or are currently engaged in deploying, some kind of automated source code analysis. The reasons for doing so can be stated in as many ways as there are people answering the question, but the basic principle can be found in all of these deployments: Tell me what's wrong with my code before I ship it - don't let me be the guy responsible for shipping a killer vulnerability or bug into the wild.

1.0 Introduction

It is planned to develop the Source Code Analyzer and translator in which we give Java or C++ source code as an input. If we give any java/c++ source code file in this translator as an input and it tells us structure of that file/program. It will tell which header files are used. What is the main class and nested classes? It also tell the inherited classes at the mode in which they are inherited, function methods, data types of the variable at the mode in which they are declared and used in the project and the scope and life time of variable such as global or local variable to particular class.

1.1 Translator Design

1.1.1 Software Design

Software was an iterative process through which requirement are translated into a blueprint for constructing the software. Initially the blue print depicts a holistic view of software. Once the software requirement was analyzed and specified, software design was first of three technical

activities- design, code generation and test that are required to build and verify the software. Each activity transforms information in a manner that ultimately results in validated computer software.

Software requirement manifested by the data, functional and behavioral models feed the design task. The design task produces a data design, an architectural design, an interface design and a component design.

Me Glaughlin [MCG91] suggest three characteristics that serve as a guide for the evaluation of good design.

- 1) The design must implement the entire explicit requirement contained in analysis model and it must accommodate the entire implicit requirement desired by end user.
- 2) The design must be readable, understandable guide for those who test and subsequently support the software.
- 3) The design should provide a complete picture of the software, addressing the data, functional behavioral domain from an implementation perspective.

1.1.2 Logical Design

Logical design gave the conceptual view of the solution. As described in the problem analysis to design a translator there was four components which were to be designed. These include:

1. Packages
2. Macros
3. Classes
4. Functions
5. Variables
6. Comments
7. Parse other file
8. Quit

1 Packages:

The translator gives the name of the packages used in the source file. The source file may be a .cpp file or a java code file for identification of the packages. The translator reads the file line by line and the occurrences of # include in the cpp file and for “import” in the java file. In translator we used the strstr () function for finding the words “#include” in cpp file and “import” in java file. The strstr () function finds the first occurrence of a substring in another string.

Syntax → char*strstr (const char*s1, const char*s2)

If it finds, then it returns a pointer to element in string 1 where string 2 begins. After that we read the name of the header file (cpp file) and name of the packages until it finds “>” character in cpp file and “;” in java file. Then it displays the name of header file and java packages using display_pack () function.

2 Macros (for cpp file only):

The translator gives the name of the macros used in the source file. The source file may be a cpp file. The translator reads the file line by line and the occurrences of #define in the cpp file. In translator, we used the strstr () function for reading the words “#define” in cpp file. The strstr () function finds the first occurrence of a substring in another string.

Syntax → char*strstr (const char*s1, const char*s2)

If it finds, then it returns a pointer to element in string 1 where string 2 begins. After that we read the name of the header file (cpp file) until it finds “>” character in cpp file. Then it displays the name of header file using display_pack () function.

3 Classes:

Class keyword was found in the same manner in which we find another keywords. After finding the class keyword it stores the name of the class in structure variable. After that for C++ file it finds “:” and for java it looks for the word “extends” and then display the name of the class and the name of the other class from which it was inherited.

4 Functions:

Main String Library functions used are:-

1. strcat → Appends one string to another

Syntax—char*strcat (char*dest, char*source)

Return Value-

Strcat returns a pointer to the concatenated strings.

2. strchr → scans a string for the first occurrence of a given character.

Syntax—char*strchr (const char*s, int c)

Return Value-

- a) On success returns a pointer to first occurrence of a character c in string s
- b) On error (if c does not occur in s0 returns NULL.

3. strcpy → copies string src to dest. Syntax—char*strcpy (char*dest, char*src)

Return Value- dest

4. strncpy → copies at most maxlen characters of src to dest.

Syntax—char*strncpy (char*dest, const char*src, int maxlen)

Return Value- dest

5. strstr → finds the first occurrence of a substring in another substring.

Syntax—char*strstr (const char*s1, const char*s2)

Return value-

- a) On success strstr returns a pointer to element in s1 where s2 begins (points to s2 in s1)
- b) On error (if s2 does not occur in s1) strstr returns NULL.

6. strtok → scans s1 for the first token not contain in s2

Syntax—char*strtok(char *s1, const char*s2)

Return Value-

The first call to strtok

- a) Returns a pointer to the first character of the first token in s1
- b) Writes a null character in to s1 immediately following the returned token

Subsequent calls with null for the first argument will work through

the string s1 until no token remains.

Other functions used:-

1. Void display_pack () :- this function is used to display the packages used in the file.
2. void display_mac ():- this function is used to display the macros used in the file
3. void display_cls ():- this function is used to display the classes used in the file.
4. void display_func ():- this function is used to display the functions used in the file
5. void_display_var ():- this function is used to display the variables used in the file
6. void display_com ():- this function is used to display the comments used in the file
7. Void trim (char*str):- trim function will take a string str as an input and it trims the starting spaces from the str.
8. void check (char*str):- this function is used to parse the line for the functions or variables.

Notations used are:

“\$”→int	“@”→char
“#”→float	“\$*”→int*
“\$**”→int**	“@**”→char*
“@**”→char**	“#**”→float*
“#**”→float**	“\$-\$”→short
“L\$”→long	“##”→double

5 Variable:

To find the variable in the source file it looks for the keyword like int, float, char, int*, int**, char*, char**, float*, short, long, double, void and then call the function check (). In which it list the name of the variables and store it in structure variable **var[var_no.].name** and display these variable name.

6 Comments:

The translator also tells the single line and multi line comments. For single line comments it finds the character // and for multi-line it finds the character /* and store the comments in a structure variable. For multi-line comments it starts from /* up to */ within these characters the written

comments are stored in a structure variable and then displayed.

7 Parse other file:

The translator also parses another file with the message:

“Enter the path of the file to be parsed:”

8 Quit:

To quit from the translator we choose 8th option.

2.0 Algorithm

Input: Files to be parsed

(File should be with extension .java or .cpp)

Output: The program produces

- 1) Names of the packages used
- 2) Names and Definitions of macros used
- 3) Names and details of the classes used. In details it will give the names of the classes with their scope from which the class is being inherited.
- 4) The signature of the functions used i.e. return type, Name, no, and type of arguments used in the function.
- 5) Numbers and names of the variables used with their respective data types
- 6) Comments given in the program with the line number at which they are defined in the program.

Step 1: Read the input file line by line.

Step 2: For each line do the following in series (i.e. one after another) mentioned below:-

- 1) Check the line for single line arguments (keyword is //)
 - a. If keystroke is present and line starts with “//” then go to step 1
 - b. If keystroke is present and line not starts with “//” then make token of the line in such a way that it will separate the comment from the line and then proceed with that token of line which does not contain the comment.

- 2) Check the line for multi-line comments (keyword is /*)
 - a. If keystroke is present and line starts with “/*” then go to step 3
 - b. If keystroke is present and line not starts with “/*” then make token of the line in such a way that it will separate the comment from the line and then proceed with that token of line which does not contain the comment.
- 3) Read next line and
 - a. Check whether closing keystrokes i.e. “/*” of comments are present or not
 - b. If these are not present then go to step 3a
Repeat the step 3 to read whole multi line comment.
- 4) Check the line for packages (keystrokes are # include, import)
 - a. If keywords are present then parse the line to get the name of the packages
- 5) Check the line for macros (e.g. keystrokes is #define)
 - a. If keystroke is present then parse the line to get the name and definition of the macro
- 6) Check the line for classes (keystroke is class)
 - a. If class keystroke is present then check the line for inherited classes (keystrokes are “:”, extends)
 - c. Parse the line to get class name and name of the inherited classes with their scope
- 7) Check the line for functions and variables (keystroke is void)

If the keystroke is present then Check the presence of the characters ‘(, ’) and not ‘;’ in the line

 - d. If these all are present parse the line to get the function return type, name and argument list

e. If these are not present then parse the line to get the variable names with their respective data type

- 8) Repeat the step 7 for all data types present i.e. for int, char, double, short, long, int*, int**, char*, char** etc.

3.0 Input

We handle each file of java and C++ in which package, classes or inheritance concept is used. Message to input the JAVA source file was displayed on the screen as follow:

“Enter the path of the file to be parsed:”

The file which was to be input to the translator must be compiled and there should be no compilation error in it or no syntax error should be there. A single syntax error or compilation time error may output wrong result or required result might not be produced.

4.0 Interface

Using the keyword interface we can fully abstract a class interface from its implementation. That is using interface, we can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables, and their methods are declared without any body.

To implement an interface, a class must create the complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism. Interface helps to understand a program. In reverse engineering, it is possible to generate alternatives views in addition to existing ones. By using these it is possible to correct existing documentations, and understand the program’s behavior better. The interface of this translator included menu bars. Each menu holds the design of each component to be designed.

5.0 Output

Output of the translator was designed in such a manner so that it can be easily understood. The output of the translator will be in the text form. The output starts from the message

“Enter the path of the file to be parsed:”

then we give the name of the file to be parsed and after that it displays the menu with different options as shown below. Then we choose the options as we want and get the desire requirements. E.g.: the translator will show the following options:

```

.....
*
*
* 1.Packages
* 2.Macros *
* 3. Classes
* 4.Functions
* 5.Variables
* 6.Comments
* 7.Parse other file *
* 8.Quit *
*
.....

```

Enter your choice (1-8)

Packages Used are:

```

-----
S.No: 1      Name: lang*.*
-----
S.No: 2      Name: java.util.*
-----
S.No: 3      Name: java.applet.*
-----
S.No: 4      Name: java.awt*
-----
S.No: 5      Name: java.awt.event.*
-----

```

Classes Used are:

```

-----
S.No: 1      Name: A
             Inherited From: nothing
-----
S.No: 2      Name: B
             Inherited From: A
-----
S.No: 3      Name: simple inheritance
             Inherited From: Nothing
-----

```

Functions Used are:

```

-----
"$" -> int      "@" -> char      "#" -> float
"$*" -> int*
"$**" -> int**   "@*" -> char*
"@**" -> char**  "#*" -> float*
"#**" -> float** "$$-->short  "L$" -> long
"##" -> double
-----

```

S.No	Return Type Arguments	Name
1	0 No Argument	showij
2	0 No Argument	showk
3	0 No Argument	sum
4	0 static void main string args[]	

Variable Used Are:

```

-----
"$" -> int      "@" -> char      "#" -> float
"$*" -> int*
"$**" -> int**   "@*" -> char*
"@**" -> char**  "#*" -> float*
"#**" -> float** "$$-->short  "L$" -> long
"##" -> double
-----

```

Data type: \$ Variable Name: i

Data type: \$ Variable Name: j

Data type: \$ Variable Name: k

Comments Used Are:

```

-----
-----
S.No: 1 At Line Number: 1

        Comment: //a simple example of
inheritance
-----
-----
S.No: 2 At Line Number: 7

        Comment: // create a super class
-----
-----
S.No: 3 At Line Number: 15

        Comment: // create a subclass by
extending class A
-----
-----
S.No: 4 At Line Number: 29

        Comment: // the super class may
be used by itself
-----
-----
S.No: 5 At Line Number: 25

        Comment: /* the sub class has
access to all public members of its super class */
-----
-----

```

5.1 Discussion

Based on the results it is shown that the Algorithm can effectively analyze and parse the source code of java and c++ but there are many parameters which can effect the solution like which source code file is used means this source code analyzer and translator can analyze and parse only .cpp and .java file.

5.1 Conclusion

Translator proposed in this research report gives the design of system, which is coded, in C++ and java library. The main items, which were abstracted from the code, were classes and their relationship, packages used in source file, different header files used, variables declared and their life and scope. The output is in a text form and is notated form. Notations are devised in translator where the main consideration about this is that the notations used are as near as UML and a UML

design can also be designed easily from our notations.

There is a lot to be done which makes the translator stronger and more active. The expectations are not handled by the translator. Try and catch blocks are not instrumented in the translator. Ways to produced information that includes exception handling should be investigated and implemented into the translator.

Interface can also help to make it more understandable and easy to use. Using the keyword interface we can fully abstract a class interface from its implementation. That is using interface, we can specify what a class must do, but not how it does it. Interfaces are syntactically similar to classes, but they lack instance variables and their methods are declared without anybody.

To implement an interface, a class must create a complete set of methods defined by the interface. However, each class is free to determine the details of its own implementation. By providing the interface keyword, java allows you to fully utilize the “one interface, multiple methods” aspect of polymorphism. Interface helps to understand a program. In reverse engineering, it is possible to generate alternative views in addition to existing ones. By using these it is possible to correct existing documentation, and understand the program behavior better. The interface for this translator included menu bars. Each menu hold the design of each component to be designed.

5.2 Future Scope of Work

There is a lot of work that can be pursued in the future like the syntax of one programming language can be converted into another language, source code of more programming languages can be analyzed and parsed.

Following improvements can be made

- Graphical user interface can be created by using C++ graphics library.
- More object oriented features can be determined like what are different abstract classes etc.

6.0 References

1. [MCG 91] McGlaughin, R., “Some notes on program design “Software Engineering

- notes, Vol 16 no. 4 October 1991, pp 53-54.
2. [DD04] Deitel H.M. and Deitel p.j. C++ how to program 4th edition. Pearson Education (Singapore) Pte. Ltd. 2004.
 3. [BROJO1] Booch Grady, Rumbaugh James and Jacobson Ivar. The unified modeling user guide-5th Indian reprint, Addison Wesley Longman(Singapore) pte. Ltd, 2001.
 4. [TR00] TIMO RAITALAAKSO, dynamic visualization of c++ programs with UML Sequence Diagrams, Master of Science Thesis.
 5. [PR01] PRESSMAN Roger S. , Software Engineering : A Practitioner approach-5th ed. McGraw-Hill series in computer science,2001.
 6. X.P. Chen, W.T. Tsai ,W.T. Tsai, J.K. Joiner, and H. Gandamaneni Lee Reynolds (Reynolds@mail.cs.umn.edu)

<http://www.hpc.unsw.edu.au>
 7. [Mu100] Muller hausi a., Understanding software systems using reverse engineering technology Research and Practice,
<http://www.rigi.csc.uvic.ca/UvicRevTut/F4rev.html>, Department of computer science, university of Victoria, 2000.
 8. L. Lavagno, G. Martin, and B. Selic. Uml for Real: Design of Embedded Real – Time Systems. Kluwer Academic Publishers, 2003.

<http://www.comp.nus.edu.sg/zhuyx/UML2SystemC.html>
 9. [KC98] Kazman Rick, Carriere Jeromy, View Extraction and view Fusion in Architectural Understanding, Proceeding of the fifth International Conference on Software Reuse (ICSR5), 1998.
 10. [Mor00] MORALE project, IS Vis tool,
http://www.gatech.edu/morale/tools/colleges_of_computing_Georgia_Institute_of_technology, 2000.