Volume 7, Issue 5, May 2017 ISSN: 2277 128X
International Journal of Advanced Research in Computer Science and Software Engineering
Research Paper
Available online at: www.ijarcsse.com

A Cyclomatic Complexity Visual Tool For Simple Source Code Coverage

Omar M. Sallabi*
Faculty of Information Technology,
University of Benghazi, Libya

Ahmad R. Benhalloum
Faculty of Arts,
University of Benghazi, Libya

Mohamed A. Hagal
Institute of Science,
University of Kastamonu, Turkey

DOI: 10.23956/ijarcsse/SV7I5/0172

Abstract— Software testing is an important part of the software development lifecycle, starting testing from initial stages allows us to avoid the difficulty of fixing bugs in advanced syages[7]. Many testing techniques have been used to discover any errors or software bugs raised during the implementation. One the of the widely used technique is the White-Box testing technique, which focuses on the structure of source code, by calculates the execution complexity of the software through the paths and control points. The tool shows visually the testing paths in a graphical way, and directs the tester to perform the test during the paths, the programmer can grantee that all pieces of the software are covered and tested. However, performs withe-Box testing manually is very difficult and requires more time and effort. In this paper, we have attempted to build an implementation tool to find test paths for any code written in Visual Basic (VB) language

Keywords— Software engineering, Testing, Cyclomatic complexity

I. INTRODUCTION

The process of software testing is an important and difficult process. One of the common techniques is the structural testing technique. The testers using this technique should analyse the unit structure (i.e. test control points, iterations and other structures) and their representation in a graphical form. These are used to compute the Cyclomatic Complexity of the code by the calculation of independent paths in the source code. These require more time and effort, as well as inadvertent errors which may occur. Testing can be classified into various stages depending on the objectives and tasks to be performed by the test at each stage of the SDLC. The test can be classified into a unit test and acceptance test [4]. The testing process consists of two parts: software configuration and configuration test. Figure 1.1 illustrates the testing information flow.

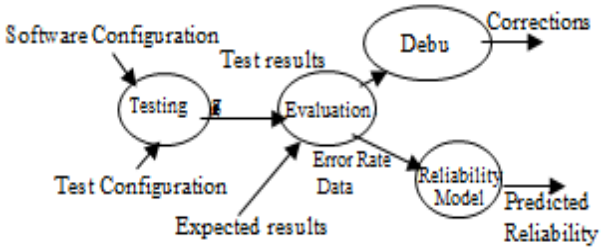

Figure 1.1. Testing Information Flow [2]

Software configuration includes requirements specification, design specification, source code and all necessary accessories. Configuration testing includes test cases, test plan, and testing tools. The evaluation process is used to compare the test results with the expected results [2].

There are two major categories of testing techniques: functional and structural. Functional testing depends on the external behaviour of a program in terms of inputs and outputs. The most famous method used in such testing is Black-box method [2]. Figure 1.2 illustrates the representation of the process of white box testing.

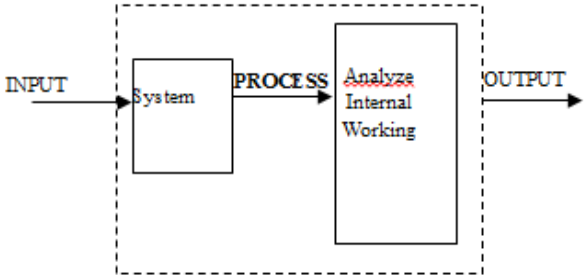

Figure 1.2. The process of White Box [1]

The tool presented in [5] has proposed a solution to the difficult manual testing to enhance the efficiency of software testing. This tool divides a system into two parts: one for reading source code, treating it and producing the executive program. The second part is for testing operation and getting results.

The java code coverage tool in [3] performed class, method, block, branch and predicate coverage. It performs the analysis at the bytecode level. "The tool uses a MySql database to store code elements and test coverage information at the granularity of individual test cases".

In addition to, some of the tools listed in [3] require special requirements such as databases and compilers of the language to compile the source code that needs to be tested.

Our motivation for developing the tool is to make the structured testing more easily; because it does not require any more requirements to be included within the given code to be tested as other tools might require. This tool has its own built-in interpreter (i.e. without the need of the compiler of the tested code).

II. THE TOOL ARCHITECTURE

In this paper, we proposed a tool assist developers to produce high-quality applications, the tool outputs is test coverage paths, which is one of the complex processed tasks for the developers. This process is the application of white-box technique on software. Figure 2.1 illustrates the tool architecture.

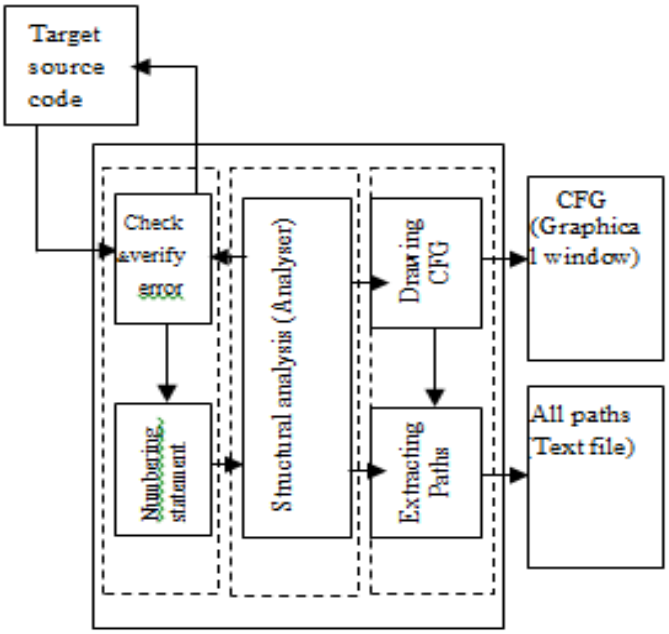

Figure 2.1. Tool architecture

The tool architecture consists of the following units:

- Input unit: the input is a text file contains the source code to be tested. The text file is analysed through a syntax analysis unit, which is checked and verified, and then each statement in the source code gives a number.
- Analysis unit: after reading and validating the input, this unit will analyse it and determines the structure of the entered code, and then it gives an order to the next unit (output unit) to draw and extract the paths.
- Output unit: This unit consists of two parts: the first part is an interpretation part which is responsible for converting the source code to the graphical representation (i.e. control flow graph(CFG)), and the second part is responsible for performs the operations related to the paths extraction.

III. BUILDING CONTROL FLOW GRAPH

Control flow graph (CFG) is used to describe the sequence of operations that will be executed in the source code[6]. The drawing of CFG is represented in the form of nodes (circles) and edges (lines). Each node will contain one or more sequence statements numbers, and each control statement (i.e. its ordered number) will be represented in one node. The edges between nodes will represent the flow from one node to another.

The conversion process requires a full understanding of the source codes of the program, in other words, understanding the structure of the program in terms of statements, reserved words, and expressions. The conversion process requires using the stack to handle the source code structure, and prepare it for the process of drawing the CFG and also extract the tracks from the analysed structure.

The analyser unit is used to analysis the source code according to the stored structure of the statements, and then draw the CFG for the source code visually in and graphical form on the screen, and then shows the extracted tracks to be tested. Figure 3.1 illustrates a simple code example for calculating the greatest common divisor (GCD) of two numbers.


```
public function GCD(x,y as integer) as integer
do while (x<>y)
if x>y then
x=x-y
else
y=y-x
end if
loop
gcd=x
end function
```

Figure 3.1. GCD function.

In this example, a stack will be used during the process of analysis and validation. "Do while" statement will be pushed into this stack, and also its drawing coordinates (x,y) will be stored in order to return to it later and continue this process until reaching the end of the loop statement. Figure 3.2 illustrates the operations.

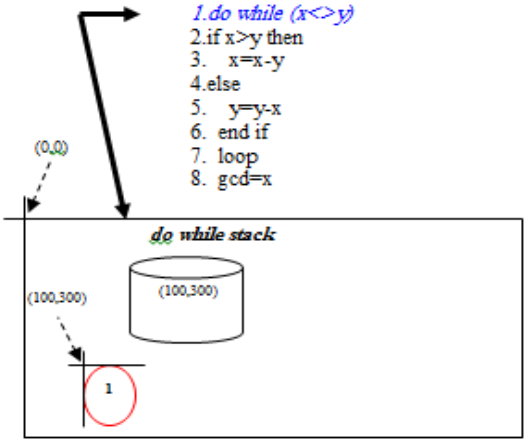

Figure 3.2: Drawing "do" while

The next statement in the source code pushed to the stack is "if statement", the tool continue reading statements until reach to either "else statement" or "endif" statement, if else statement two pushes to the stack are required. (see figures 3.3 and 3.4).

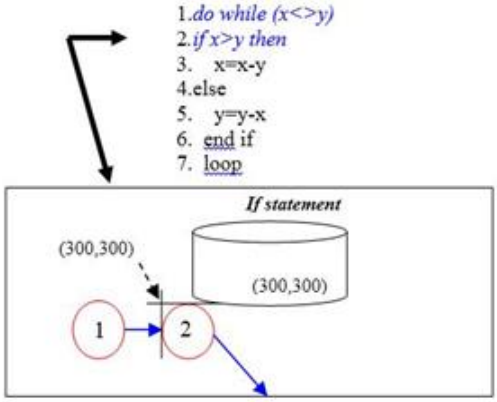

Figure 3.3: Drawing "if" statement

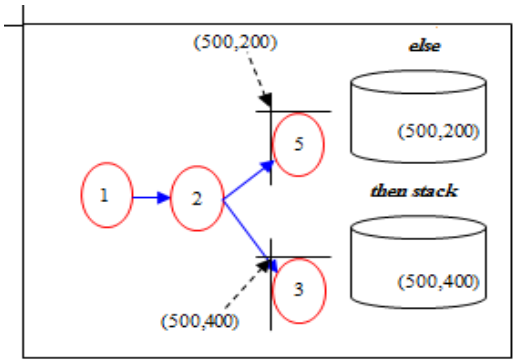

Figure 3.4: Drawing "else" statement

When reaching the "endif" statement, a test operation is required to check whether the statement ("if" statement) contains "else" statement or not. If contained "else" statement, the tool will draw its node. Figure 3.5 illustrates such operation.

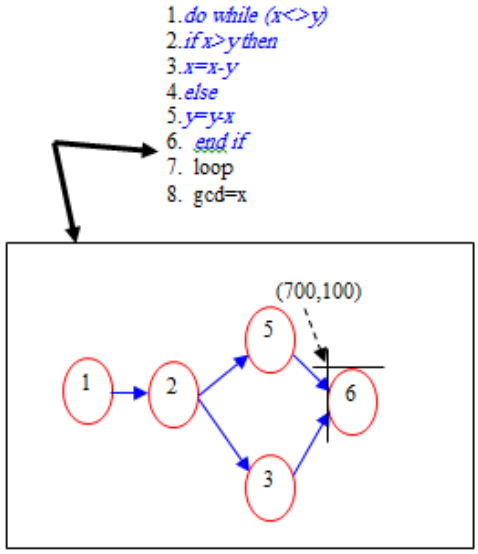

Figure 3.5. Drawing "endif" statement

Finally, reading the "Loop" statement, and then reading the "do while" statement with its position in the drawing. After that, the tool starts drawing the node which represents the "loop" statement and connecting with the edge which represents the "do while" iteration in an opposite direction. Also, drawing the edge represents the exit from the iteration or in the case of the unsatisfied condition of entrance to the "loop" statement which is the next statement after "loop" statement.

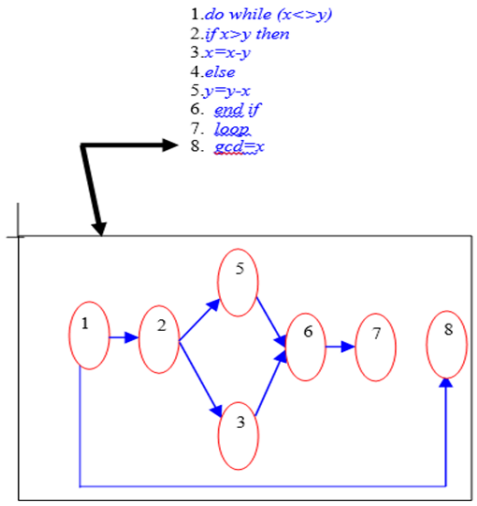

Figure 3.6. Drawing "loop" statement.

IV. CALCULATION OF THE CYCLOMATIC COMPLEXITY AND PATH EXTRACTION

The main objectives of the tool are the extraction of execution paths in the source code. This operation will help the testers in identifying the number of paths and the calculation of edges and nodes inside the CFG.

McCabe's Cyclomatic complexity able to define and identify the maximum number of the individual paths within the program identifies these methods mathematically (i.e. simple mathematical operations). There are also further different methods for calculating the McCabe's Cyclomatic complexity. Each of these methods has a certain calculation approach depend on some elements in the CFG; such as the number of nodes, edges or the control points [4].

In our tool the extraction of paths depending on CFG, we will use the structural edges and nodes shown earlier, and build a new structure for the integration of these two structures using a table, which is a two-dimensional array and its size equal the number of edges in the CFG, and then we can analyse this array and extract the paths , each node has entered edges and out edges. This represents the transition between the nodes. Hence, we compare each node (depending on its location, height, and width) with the start and the end of each edge (start and end points). If the beginning of the edge is located within the area or the vicinity of the node (i.e. this edge entered for this node), then this node will be linked to the node that it touches. The same way is done with the edge that is on the other side. Figure 4.1 illustrates the relationship between the nodes and their linked edges.

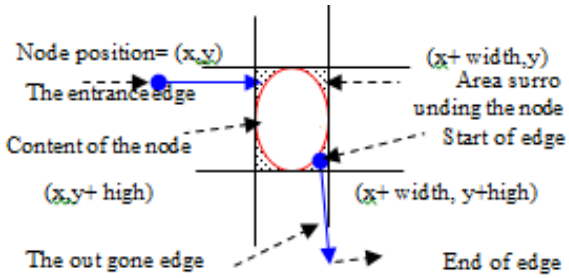

Figure 4.1. The relationship between the nodes and linked edges to them.

After applying the analysis process, we will extract all the independent paths within the source code by using the CFG. During the analysis process, an array will be created and filled with all the data, which contains the drawing position of each node and edge of the CFG. Also, another array will be created contains the contents of each node (i.e. the code's statements numbers of the given code (see figure 3.6). Table 4.1 represents the drawing position of each node and edge in the graph, and table 4.2 represents the contents of each node.

Table 4.1. Graphic data derived from CFG diagrams

Node number	Node Position	Edge	
		Start Position	End Position
1	(100,100)	(120,120)	(150,120)
2	(150,100)	(170,120)	(200,150)
3	(200,150)	(170,120)	(200,50)
4	(200,050)	(220,50)	(250,120)
5	(250,100)	(220,150)	(250,120)
6	(300,100)	(270,120)	(300,120)
7	(350,100)	(270,120)	(300,120)

Table 4.2. The contents of every node

Node #	1	2	3	4	5	7	7
Contents	1	2	3	5	6	7	8

For example, from the table (table 4.1) above, we note that the starting points of the edge #1 trapped within the area of node #1. This means the edge is going out from the node #1. Then, retrieving the node that touches the edge #1 from the other side (end points); here is a node #2. In this example, we can conclude that the node #1 is associated with the node #2 through the edge #1. The same operation applied to the other nodes and edges.

Finally, these results will be represented in the new two-dimensional array (its size (n X n) where n is the number of nodes). The initial values of this array are zero. This array represents the traceability of the transitions between the nodes. For example, if node #1 is linked with node #2, this cell which represents the intersection between them will be filled by the value "1". Table 4.3 illustrates the generated array that represents all the nodes and their linked nodes.

Table 4.3. The generated array after matching nodes with each other.

Node number		**Destination node**						
		1	**2**	**3**	**4**	**5**	**6**	**7**
Source node	**1**	**0**	**1**	**0**	**0**	**0**	**0**	**1**
	2	**0**	**0**	**1**	**1**	**0**	**0**	**0**
	3	**0**	**0**	**0**	**0**	**1**	**0**	**0**
	4	**0**	**0**	**0**	**0**	**1**	**0**	**0**
	5	**0**	**0**	**0**	**0**	**0**	**1**	**0**
	6	**1**	**0**	**0**	**0**	**0**	**0**	**0**
	7	**0**	**0**	**0**	**0**	**0**	**0**	**0**

V. CONCLUSION

In this paper, a tool for structured testing is presented. Our tool simplifies the testing operation by analysing a given source code (written in VB language) by presenting its CFG as well as the independent (execution) paths in the source contains.

The tool will read the source code without any further requirements such as the compiler, the tool has its own interpreter. The tools can be used within the academic curricula for the software testing course, which helps students to increase their understanding of path tracing and how can they benefit from using the technique of White Box Testing in getting curated system results.

Also, we could conclude that time and effort taken manually by software tester will be reduced as well as its reliability and accuracy degrees also increased.

REFERENCES

[1] Mohd.Ehmer Khan, "Different Forms of Software Testing Techniques for Finding Error", International Journal of Computer Science Issues, Vol. 7, Issue 3, No 1, 2010.

[2] LuLuo., " Software Testing Techniques Report", School of Computer Science, CarnegieMellon University, USA, 2010.

[3] Raghu, Lingampally, Gupta Atul and JalotePankaj, " A multipurpose Code Coverage Tool for Java", 40thHawaii International Conference on System Sciences, India-Kanpur, 2011.

[4] Shah, Amrish, Ajitha and etl, " Software Testing Guide Book: Fundamentals of Software Testing, Free Software Foundation.", 2004.

[5] Xue, Ying Ma and Kui Shengbin, " Designing Test Engine for Computer-Aided Software Testing Tools", WSEAS TRANSACTIONS on COMPUTERS 10.5, ISSN 1109-2750, 2011.

[6] Ammann, Paul and Jeff Offutt, " Introduction to Software Testing", New York: Cambridge UP, 2008.

[7] T.Rajani Devi,“ Importance of Testing in Software Development Life Cycle”, International Journal of Scientific & Engineering Research Volume 3, Issue 5, May-2012 1 ISSN 2229-5518

