Development Of Testable Paths For Java Programs

Thida Lwin, Swe Swe Shein

*Computer University (Taunggy*i)

thidalwin.ucsm@gmail.com, ssksucsy@gmail.com,

Abstract

Software testing is a critical element of software quality assurance. One of the important tasks during software testing is the generation of testable paths. Software testing is extremely labor and resource intensive, accounting for 50-60% of the total cost of software development. Testing is integrated with and affects all stages of the Software Engineering lifecycle. This system has code parsing module from input java source code, analyzer module which maps with predefined keywords, control flow graph generator and developing testable paths module. This system calculate cyclomatic complexity to validate the output testable paths. This system evaluates the testable paths of the input Java source code and generates the control flow graph (CFG) to assist the software developer in testing phase.

Keywords: Software Testing, path testing, class analyzer, parser.

1. Introduction

Software testing means the process of analyzing a software item to detect the difference between existing and required conditions (i.e., bugs) and to evaluate the features of the software item. Testing techniques can be broadly classified into two categories, functional and than implementation of the program. Structural testing is concerned with the use of the control-flow a program to guide the generation of testable paths.

An important technique is usually concerned with the use of the control-flow of a program. To adequately test the program at the structural level, structural elements (nodes, branches, or paths) of the control flow graph of coverage must be considered. Statement coverage requires developing test path to execute certain nodes of the Control Flow Graph (CFG).

The Java approach to software development does not rule out the need to test the software. The Java paradigm is successfully applied in many software projects, and the use of Java languages is increasingly widespread. Java technology has been widely studied and applied, due to its advantages in improving the productivity and reliability in software development. Programs developed with Java technologies have unique features that often make traditional testing methods inadequate. Several techniques proposed in the literature for testing Java software and investigate the impact of Java approach on the design of testing strategies. Little attention has been paid to testing of Java programs[1].

1.2. Motivation

After initial testing, programmers face problem of finding additional test data to evaluate program elements not yet covered. Finding input data to evaluate those remaining elements requires a good understanding of the program under test from the programmers and can be very labor intensive and expensive.

To test a program, it is necessary to generate testable paths from the input domain of the program under test. Therefore, a key issue in software testing is how to generate adequate test paths from the program input domain to detect as many faults as possible with a minimum cost. If paths could be automatically generated, the cost of software testing would be significantly reduced.

The main motivation is to help programmer in software testing. One particular technique that has been suggested to aid programmers in software testing is path testing. Path testing can be used to assist the programmer in a lot of tedious and defect faults tasks software testing and software quality assurance. In this paper, we

focus on the approach to generate testable paths for Java program.

1.3. Related Work

The paper[7] present a method for testing computer programs with iteration loops and identifies as sequences of simple loop paths. A software tool called SILOP has been developed to automatically generate these simple loop patterns and each corresponding sequence of simple loop paths can be considered as a test case. This paper[9] presents a novel method of BPEL test case generation, which is based on concurrent path analysis. This method uses an Extended Control Flow Graph (XCFG) to represent a BPEL program, and generates all the sequential test paths from XCFG. These sequential test paths are then combined to form concurrent test paths. Finally a constraint solver is used to solve the constraints of the setest paths and generate feasible test cases. Other work[2] focuses on the combination of functional and structural testing, but does not tackle the problem of generating structural-based testable paths. Some test generation methods[10][5] based on path analysis are also proposed for testing sequential and concurrent programs. These methods firstly select local paths for individual tasks, then compose global paths with these local paths. This paper[11] proposes a graph-search based approach to BPEL test case generation, which effectively deals with BPEL special features. This approach defines an extension of CFG (Control Flow Graph) - BPEL Flow Graph (BFG) - to represent a BPEL program in a graphical model. Then concurrent test paths can be generated by traversing the BFG model, and test data for each path can be generated using a constraint solving method.

2. Background Theory

2.1 Software Testing

Software testing is an expensive and difficult process which need much time. The objective of software testing is to detect faults in the program and therefore provide more assurance for the quality of the software. Testing is one of the vital activities of software development. As an important stage to guarantee software quality and reliability, software testing plays an irreplaceable role in the process of software development.

It is the one of the important method of software quality, reliability, and safety assurance. Software testing should focus on interaction between the components and on the functionality and performance of the system as a whole. Software testing, as crucial part of quality assurance, should also focus on bug prevention. The testing procedure consists of selecting elements from the program's input domain, executing the program on these test cases, and comparing the actual output with the expected output. The pragmatic goal of software testing is to discover bugs-discover symptoms caused by bugs, and provide clear diagnoses so that bugs can be easily corrected[6].

There are two distinct types of testing[1] that may be used at different stages in the software process:

1. *Defect testing* is intended to find inconsistencies between a program and its specification. These inconsistencies are usually due to program faults or defects.

2. *Statistical testing* is used to test the program's performance and reliability and to check how it works under operational conditions.

2.2. Defect Testing

The goal of defect testing [1] is to expose latent defect in a software system before the system is delivered. A successful defect test is test which causes the system to perform incorrectly and hence exposes a defect.

There are several techniques for defect testing:

- **Black-box Testing**

The system is black-box whose behavior can only be determined by studying its inputs and the related outputs.

- **Equivalence Partition**

One systematic approach to defect testing is based on identifying all equivalence partition which must be handled by a program.

- **Structural Testing**

The analysis of the code can be used to find how many test cases are needed to guarantee that all of the statements in the program or component are executed at least once during the testing process.

- **Path Testing**

Path testing is a structural testing strategy whose objectives is to exercise independent execution path through a component or program.

2.3. Path Testing

Path testing is a structural testing strategy whose objectives is to exercise independent execution path through a component or program. If every independent path is executed then all statements in the component must have been executed at least once. Furthermore, all conditional statements are tested for both true and false case. As modules are integrated into systems, it becomes unfeasible to use structural testing techniques. Path testing techniques are therefore mostly used at the unit testing and module testing stages of the testing process. Path testing strategies involve selecting test data on the basis of which program path or paths are to be exercised[3].

The starting point for path testing is a program flow graph. This is a skeletal model of all paths through the program. Second, the number of independent paths in a program can be discovered by computing the cyclomatic complexity of the program. After discovering the number of independent paths through the code by computing the cyclomatic complexity, the next step is to design test case to execute each of these paths.

Path testing techniques are usually concerned with the use of the control-flow to guide the generation of path. Path testing based conceptually on the representation of a program as a flow graph (CFG)[4]. Statement coverage requires developing test cases to execute certain nodes of the CFG. Similarly, branch coverage requires test cases to traverse certain branches, and path coverage requires test cases to execute certain paths. Path testing thus includes (i) choice of a criterion (statement, branch or path), (ii) identification of a set of nodes, branches or paths of this set[8].

2.4. Cyclomatic Complexity

Cyclomatic complexity measures the amount of decision logic in a single software module. It is used for two related purposes in the structured testing methodology. First, it gives the number of recommended tests for software. Second, it is used during all phases of the software lifecycle, beginning with design, to keep software reliable, testable, and manageable. Cyclomatic complexity is based entirely on the structure of software's control flow graph. Cyclomatic complexity is defined for each module to be

$$V(G) = e - n + 2$$

where; e and n are the number of edges and nodes in the control flow graph, respectively. Cyclomatic complexity is also known as v(G), where v refers to the cyclomatic number in graph theory and G indicates that the complexity is a function of the graph[2].

2.5. An Example of Path Testing

```
public void BinSearch(int key, int[] Array, Result r)
{    while(bottom<=top)
     {  mid=(top+bottom)/2;
        if(Array[mid]==key)
        {    r.index=mid;
             r.found=true;
             return 0; }
        else
         {  if (Array[mid]<key)
              bottom=mid+1;
              else top=mid-1; }
}   }
```

Figure1: Java implementation of a binary search routine

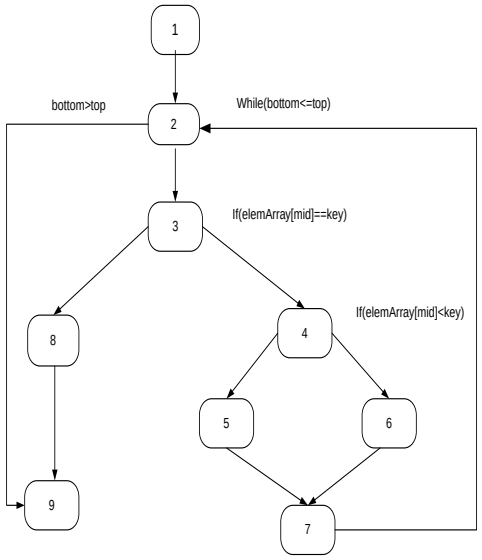

Figure 1. Flow graph for a binary search routine

By this figure the independent paths for testing are;

- 1,2,3,8,9
- 1,2,3,4,6,7,2
- 1,2,3,4,5,7,2
- 1,2,3,4,6,7,2,8,9

If all of these paths are executed can be sure that

- Every statement in the method has been executed at least once, and
- Every branch has been exercised for true and false condition.

Cyclomatic complexity for control flow graph is

$$V(G)=10-8+2=4. \quad (1.1)$$

Therefore, program paths and cyclomatic complexity is equal and this is provided to test software tester.

3. Overview System Architecture

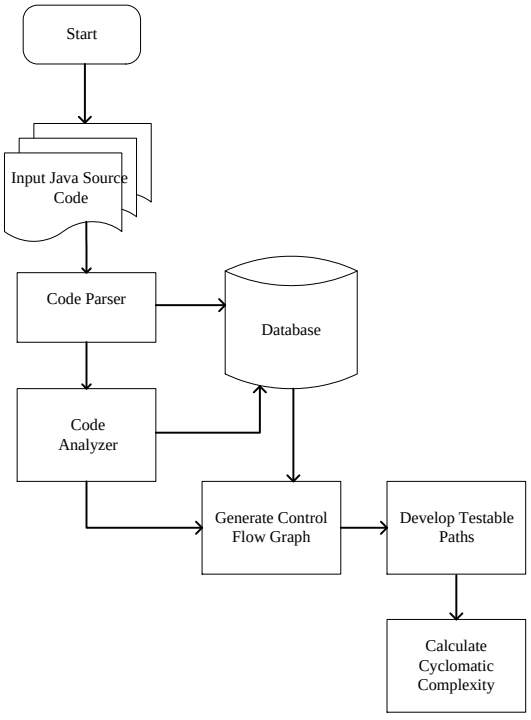

Figure 2. Overview of the system architecture

This section explains the process of the generation of testable paths for java source code. The main components cooperating to this system are java source code parser, code analyzer, control flow graph (CFG) generator, testable path illustrator and cyclomatic complexity measurement.

(i) Java Source Code Parser

This system emphasizes to evaluate the testable paths focus on the control words such as 'For', 'While', 'do While', 'If', 'Else if', 'Else'.

This module scans the input java source code line by line, skip the blank lines and comment lines and stores into code table. Limitation for this module is the input java source code must be written in standard format.

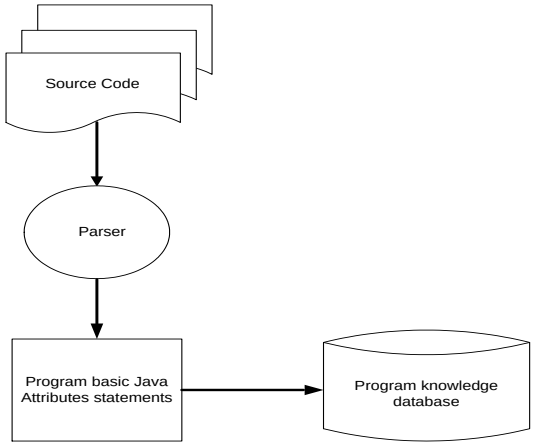

Figure 3. Source Code Parser

(ii) Code Analyzer

```
1       while
2       {
3               if
4               {
5               }
6               else if
7               {
8                       If
9                       {
10                      }
11                      else if
12                      {
13                      }
14              }
15              else
16              {
17              }
18      }
```

Table 1. Sample Code Table

This module analyzes the code table to fetch control words, scope words '{', '}' and, filters unnecessary statements and instructions, within block {}, for this module and modifies the code table. Code analyzer causes the code table which records pure control words and scope words {} as shown in table 1.

(iii) Control Flow Graph Generator

In this phase, complex data structure is needed to retain the track of the program. This module defines a structure named with 3 main attritutes called LinkNode as follows.

```
LinkNode
(
        LinkNode * Parent,
    String NodeNo,
    String NodeLabel,
    Vector<LinkNode *> Child
)
```

Parent is the root/previous node of node and child is leaves/next node of current. NodeNo is the position and NodeLabel is the name of current and which may be control keywords (while, for, if, else-if, else).

This module first scans the code table shown in Table 1 and generate the linked LinkNodes, which is the critical part and it will be source to generate the control flow graph and also useful for testable path illustration.

Algorithm for Linked LinkNode Creation

```
While (not end of code table)
{   Create root node firstly
    Scan the table line by line
    If  found word if keyword
        Create_Linked_LinkNode(keyword)
        {   Create node for keyword
                While it has nested keyword
            Create_Linked_linkNode(keyword)
        }
}
```

In this algorithm, to know the start '{' and end '}' of code block, stack is used. While the code table is scanned any keyword 'for', 'while', 'if', 'else-if', 'if', '{' and '}', these are push into the stack and pop it whenever the node is created for any keyword.

By traversing the linked LinkNode, the control flow graph is generated as shown in Figure 4.

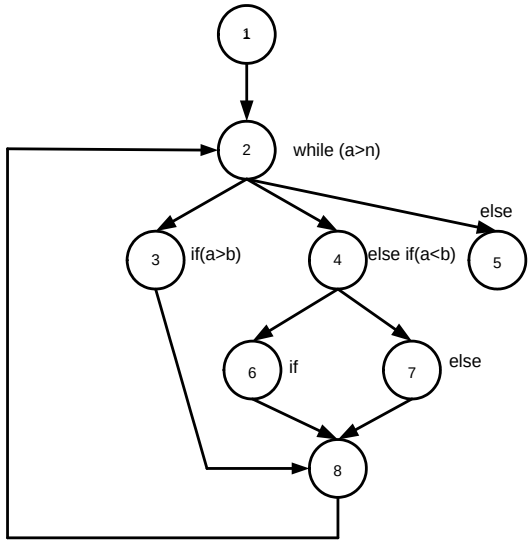

Figure 4. Control Flow Graph For Table 1

(iv) Testable Path Illustrator

While traversing the linked LinkNode, not only generate the control flow graph but also record the testable in paths table and the following testable paths are illustrated.

- **Path 1= 1,2,3,8,2**
- **Path 2= 1,2,4,6,8,2**
- **Path 3=1,2,4,7,8,2**
- **Path 4=1,2,5**

(v) Measuring Cyclomatic Complexity

To validate the number of testable paths are consistent, this module calculate the cyclomatic complexity for given source code by using equation (1.1).

V(G) = no. of Edge – no. of Node + 2

V(G) = 10 – 8 + 2 = 4

According to the above calculation, the number of testable paths is equal to the program generated testable paths.

14. Conclusions

Software testing is important and expensive. Path testing is one of the important techniques of software testing. This system generates the program flow graph for Java-based program. This system introduces an approach to develop the testable program paths from program flow graph. This system intends to support in performing the procedure of the path testing.

15. References

[1] B. Beizer, "Software Testing Techniques", International Thomas Publishing Inc., 2nd Edition, 1990.

[2] M. Cabe, J. Thomas, Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric, 1982. NIST Special Publication 500-99, Washington D.C.

[3] Clarke, A. Lori, A Formal Evaluation of Data Flow Path Selection Criteria, 1989.

[4] D. Hoffman and P. Strooper, "Graph-based classes testing", Procedding of the 7th Austmlian Software Engineering Conference, September 1993.

[5] T. Katayama, E. Itoh, and Z. Furukawa, "Test-case generation for concurrent programs with the testing criteria using interaction sequences", Proceedings of the 6th Asian-Pacific Software Engineering Conference, December 1999, pp. 590-597.

[6] P. Roger, Software Engineering: A Practitioner's Approach, 5th Edition, 2001. McGraw Hill, Boston.

[7] J. L. White, W. Bogdan, "Path Testing of Computer Programs with Loops using a Tool for Simple Loop Pattern", Software-Practice and Experience, Vol. 20(10), October 1991,pp-1075-1102.

[8] M.R.Woodward, D.Hedley,and M.A.Hennell " Experience with path analysis and testing of programs ", IEEE Transactions of Software Engineering.

[9] J. Yan, Z. Li, Y. Yuan, W. Sun, J. Zhang, "BREL$WS Unit Testing: Test Case Generation Using a Concurrent Path Analysis Approach", 17th International Symposium on Software Reliability Engineering (ISSRE'06), May 2006.

[10] R. D. Yang and C. G. Chung, "A path analysis approach to concurrent program testing", Information and Software Technology, 1992, pp.

[11] Y. Yuan, Z. Li, W. Sun, "A Graph-search Based Approach to BPEL4WS Test Generation", Proceedings of the International Conference on Software Engineering Advances (ICSEA'06), May 2006.