TEST SCENARIOS GENERATION USING PATH COVERAGE

Vikas Panthi
National Institute of Technology Rourkela
14 PUBLICATIONS **53** CITATIONS
SEE PROFILE

Durga Prasad Mohapatra
National Institute of Technology Rourkela
154 PUBLICATIONS **509** CITATIONS
SEE PROFILE

TEST SCENARIOS GENERATION USING PATH COVERAGE

VIKAS PANTHI[1], D. P. MOHAPATRA[2]

[1,2]Dept. of Computer science & Engineering, National Institute of Technology, Rourkela, Orissa, India 769008
E-mail: vpanthi@india.com, durga@nitrkl.ac.in

Abstract- Testing is one of the very important component of software development process. Properly generated test sequences may not only locate the defects in software, but also help in reducing the high cost associated with software testing. It is often desired that test sequences should be automatically generated to achieve required test coverage. Automatic test sequence generation is a major problem in software testing. The aim of this study is to generate test sequences for source code using ModelJunit. ModelJUnit is a extended library of JUnit. We have generate automatic test sequences and some testing criterion coverage such as node coverage, edge coverage and edge pair coverage. This paper describes a systematic test sequence generation technique using the path based approach.

I. INTRODUCTION

Software testing is the process of executing a program with the intent of finding errors. Software testing is essentially a combination of software validation and verification. Software's quality is closely related to our lives. The amount of software in general consumer products, is doubling every two to three years [3]. As complex programs become integrated into all aspects of society, it is important that there exist no errors that could compromise safety, security or even financial in-vestment. Random testing, constraint-based testing and model checking based testing methods are helpful for automatically generating test cases from the software system. Software testing can be classified in two types: functional and structural testing. Functional testing is basically used for testing the functionality of the system using some functional test criteria. Some example of functional testing are Boundary value testing, Decision table based testing, Random testing, Equivalence testing etc. Structural testing is used for the identified structural of the system using some structural test criteria such as paths, functions, conditions, branches, Data flow testing etc. The discussed characteristic of structural testing methods is that they are all based on the source code, not on the specification. Because of this absolute basis, structural testing methods are very amenable to rigorous definitions, mathematical analysis, and precise measurement. Basis path is one of the very powerful structural testing criteria. Basis path testing requires that, the number of test paths (basis set) should be equals to the cyclometic complexity of program [1] such that every path is an independent path and all edges in the control flow graph (CFG) are covered by all the paths in the basis set. In addition, a path that is not contained in the basis set can be constructed by a linear combination of paths in this set. Zhonglin and Lingxia [6] and Qingfeng and Xiao [7] used cyclomatic complexity for generating a sequence of linearly independent paths. There are many basis paths which are infeasible because there are data dependency exist in decision node. They combined the baseline method with the dependency relationship to avoid selecting infeasible paths. These methods did'nt handle loops. Bint and Site [10] presented a path generation method based on genetic algorithm. The disadvantage of this method is that the generated set of paths cannot cover all edges of the CFG because the loop operation is removed. This method cannot generate a basis set of test paths. Guangmei et al. [4] discussed an automatic generation method of basis set of paths which is built by searching the CFG by depth-first searching method. In order to avoid that the algorithm will never stop, and for reducing the searching procedure, the sub-path from the multi-indegree nodes to the end node of a program and the sub-path that contains a loop are recorded during the construction of a basis path. This method did not handle infeasible paths. In this paper, we specifically examine path testing and generate test sequences for source code using ModelJunit. By using source code, we generate test sequences automatically and achieves some test coverages such as node coverage, edges coverage and edges pair coverage. The rest of the paper is organized as follows. Section 2 introduces MacCabe's Basis Path with CFG. Section 3 gives Basic definitions. Section 4 presents proposed Algorithm. Section 5 introduces a case study of the proposed method. Section 6 concludes the paper and future work.

II. DEFINITIONS AND BASIC CONCEPTS

In this section, first we present some definitions and then discuss basic concepts which will be required for understanding our proposed work. Definition 1 Test Case: A test case is the triplet [I, S, O], where I is the initial state of the system at which the test data is input, S is the test data input to the system and O is the expected output of the system [from our paper]. The output produced by the execution of the software with a particular test case provides a specification of the actual software behavior.

Definition 2 Path: The number of predecessors of a node is its in-degree, and the number of successors of the node is its out-degree. A path from a node x_1 to a node x_n in a graph G = (V, E) is a sequence of nodes $(x_1, x_2 ..., x_n)$ such that (x_i, x_{i+1}) E for every i, $1 \leq i \leq n-1$.

Definition 3 Subpath: A subpath P from vertices n_i to n_k is a sequence of nodes $n_i, n_{i+1}, ..., n_k$, where for each adjacent pair of nodes $(n_{i+j}\ ,\ n_{i+j+1})$ there is an edge in G for 0 <i <k I [my paper IJCA].

Definition 4 DD-Paths: A decision-to-decision (DD-Path) is sequence of nodes in a program graph such that:

- case 1: It consist of a single node with indeg = 0.
- case 2: It consist of a single node with outdeg = 0.
- case 3: It consist of a single node with indeg $\geq$ 2 or outdeg $\geq$ 2.
- case 4: It consist of a single node with indeg = 1 and outdeg = 1.
- case 5: It is a maximal chain of length $\geq$ 1.

Definition 5 Path Coverage Criterion:In this section, we discuss some of the relevant coverage criteria which are used in our approach.

a) Node Coverage: It covers every node in control flow graph for basic test sequence generation. Node coverage is a test adequacy criterion that requires tests to check programs output variables [6]. All variables still defined when executing in test scope (even those which are not visible, such as private fields of objects) are considered by node coverage.

b) Edges Coverage: A test set TS is said to achieve edge path coverage if given a control flow graph G, TS causes each possible edge in G to be taken at least once [3].

Definition 6 Extended Finite State Machine (EFSM): An Extended Finite State Machine (EFSM) is defined as a 7 tuples

M = (I, O, S, D, F, U, T) Where

I = set of input symbols.

O = set of output symbols.

S = Set of symbolic states.

D= an n-dimensional linear space $D_1 \times D_2 \times D_n$.

F= set of enabling functions fi: D $\rightarrow$ {0, 1}.

U= is a set of update functions U_i: $D_i \rightarrow D_j$

T= Transition relation T: S ×F ×I $\rightarrow$ S ×U ×O

II. MCCABE'S BASIS PATH WITH CFG

A. Control Flow Graph:

The control flow graph Gf = (N,E) of a function f has one node na N for each statement a in f and two additional nodes nin; nout. We add an edge (na, na) if the statement a is executed immediately after the statement a. For the first statement a1 in the function, we introduce an edge (nin; na1). Furthermore, we add edges (na, nout) for each node na that is associated to a statement a, after which the control flow leaves the function because of a return-statement or the right brace that terminates the function. The control flow graph of an empty function, i.e., a function without any statements consists of N = nin, nout and E = (nin; nout).The control flow graph of a program P is a connected oriented graph composed of a set of vertices, a set of edges and two distinguished nodes, e the unique entry node, and s the unique exit node. Each node represents a basic block and each edge represents a possible branching between two basic blocks. The program graph is a directed graph in which nodes are statement fragments and edges represent flow of control. If i and j are nodes in the program graph, an edge exists form node i to node j if the statement fragments corresponding to node j can be executed immediately after the statement fragment corresponding to node i. It is illustrated here with program of the quadratic equation. The importance of the program graph is that program executions correspond to paths from the source to the sink nodes. because test cases force the execution of some such program path, we now have a very explicit description of the relationship between a test case and the part of the program it exercises.

B. McCabe's Cyclometic Complexity:

Directed graph that we might take to be the program graph of some program.

Mathematically, it is set of independent paths through the graph diagram. The complexity of the program can be defined as V(G) = E –N + 2

Where,

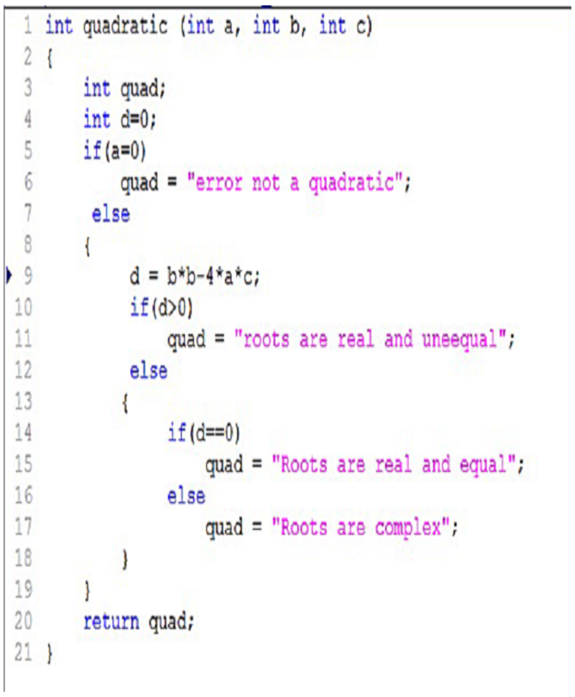
E = Number of edges

N = Number of Nodes

V (G) = P + 1

Where P = Number of predicate nodes (node that contains condition)

To illustrate the complexity that arises when branches issue in branch delay slots, consider the following code fragment: A native scheduling of this code for single delay slot architecture produces the following control flow graph:



```
int quadratic (int a, int b, int c)
{
    int quad;
    int d=0;
    if(a=0)
        quad = "error not a quadratic";
     else
    {
         d = b*b-4*a*c;
         if(d>0)
             quad = "roots are real and uneequal";
         else
        {
             if(d==0)
                 quad = "Roots are real and equal";
             else
                 quad = "Roots are complex";
        }
    }
    return quad;
}
```

Figure 1. A simple example for CFG.

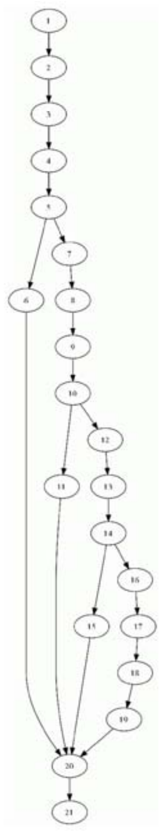


Figure 2. Control Flow Graph for the example Program in Fig. 1

Computing mathematically V(G) = 23 −21 + 2 = 4
V(G) = 2 + 2 = 4
Basis Set A set of possible execution path of a program

- 1→2→3→4→5→6→20→21.
- 1→2→3→4→5→7→8→9→10→11→20→21.
- 1→2→3→4→5→7→8→9→10→12→13→14→15→20→21.
- 1→2→3→4→5→7→8→9→10→12→13→14→16→17→18 →19→20→21.

Properties of Cyclomatic complexity:
Following are the properties of Cyclomatic complexity:

- V (G) is the maximum number of independent paths in the graph.
- V (G)$\geq$ 1.
- G will have one path if V (G) = 1.
- Minimize complexity to 10.

IV. PROPOSED ALGORITHM

In this section we, discuss our proposed approach automatically generating test sequences using path coverage. Our approach for generating test sequences the steps schematically given below:

Step 1: Develop the algorithm or source code of problem.

Step 2: According to DD-paths rules draw a control flow graph.

Step 3: Remove sequences and parallel redundancy from Control flow graph according to DD-paths rules.

Step 4: Construct the mapping table.

Step 5: Again follow the rules of DD-paths according to step 3 for removing the sequences and parallel redundancy.

Step 6: Write Java source code for using the above graph& include Modeljunit.jar library.

step 7: Generate test sequences of graph.

V. CASE STUDY

Triangle problem is the most widely used example in software testing. The triangle program accepts three integers a, b, and c as input. These are taken to be sides of a triangle. The triangle program which accepts three integers a, b and c as input must satisfy the following conditions:

c_1: 1$\leq$ a $\leq$ 200 c_4: a $<$ b + c
c_2: 1 $\leq$ b $\leq$ 200 c_5: b $<$ a + c
c_3: 1 $\leq$ c $\leq$ 200 c_6: c $<$ a + b

The output of the program is the type of triangle, determined by the three sides-equilateral, Isosceles, scalene, or Not a Triangle.

If an input failed any of the conditions c_1, c_2 or c_3, the program notes this with an output message. For example Values of a, b and c which the satisfy conditions c_1, c_2 and c_3 are four mutually exclusive output. They are given below:

1. If all three sides are equal. The program output is equilateral.
2. If exactly one pair of side is equal, the program output is isosceles.
3. If no pair of sides is equal, the program output is scalene.
4. If any of conditions c_4, c_5 and c_6 is not satisfied, the program output is Not a triangle.

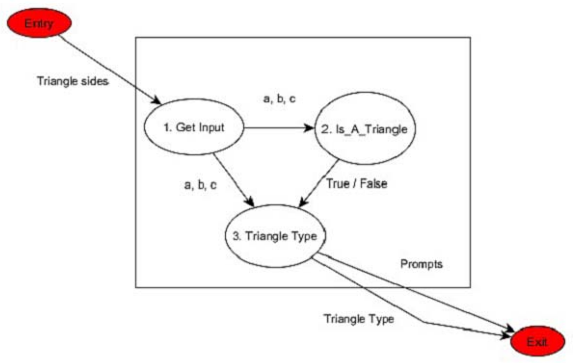
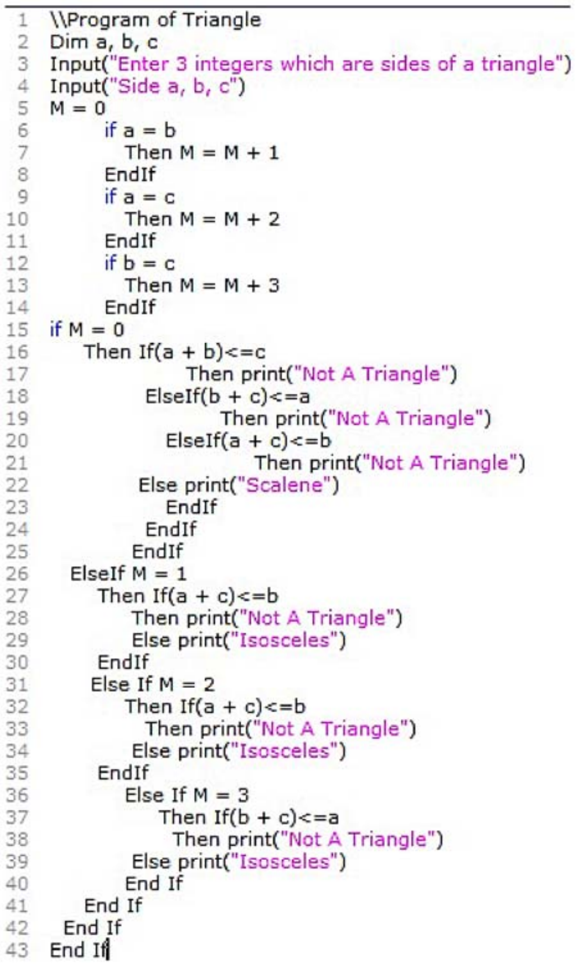


Figure 3. Dataflow diagram for a structured triangle program.

The variable "match" is used to record equality among pairsof the sides. If two sides are equal, say, a and c, it is only necessary to compare a + c with b. (Because b must be greater than zero, a + b must be greater than c, because c equals a.) This observation clearly reduces the number of comparisons that must be made.

ModelJUnit is a Java library that extends JUnit to support model based testing. Models are extended finite state machines that are written in a familiar and expressive language: Java. ModelJUnit is an open source tool, released under the GNU GPL license. ModelJUnit allows you to write simple finite state machine (FSM) models or extended finite state machine (EFSM) models as Java classes, then generate tests from those models and measure various model coverage metrics.



```
1  \\Program of Triangle
2  Dim a, b, c
3  Input("Enter 3 integers which are sides of a triangle")
4  Input("Side a, b, c")
5  M = 0
6          if a = b
7             Then M = M + 1
8          EndIf
9          if a = c
10            Then M = M + 2
11         EndIf
12         if b = c
13            Then M = M + 3
14         EndIf
15 if M = 0
16      Then If(a + b)<=c
17                 Then print("Not A Triangle")
18           ElseIf(b + c)<=a
19                 Then print("Not A Triangle")
20              ElseIf(a + c)<=b
21                    Then print("Not A Triangle")
22          Else print("Scalene")
23              EndIf
24           EndIf
25         EndIf
26    ElseIf M = 1
27       Then If(a + c)<=b
28           Then print("Not A Triangle")
29           Else print("Isosceles")
30       EndIf
31      Else If M = 2
32          Then If(a + c)<=b
33             Then print("Not A Triangle")
34           Else print("Isosceles")
35       EndIf
36          Else If M = 3
37              Then If(b + c)<=a
38                Then print("Not A Triangle")
39           Else print("Isosceles")
40          End If
41      End If
42    End If
43 End If
```

Figure 4. A sample program.

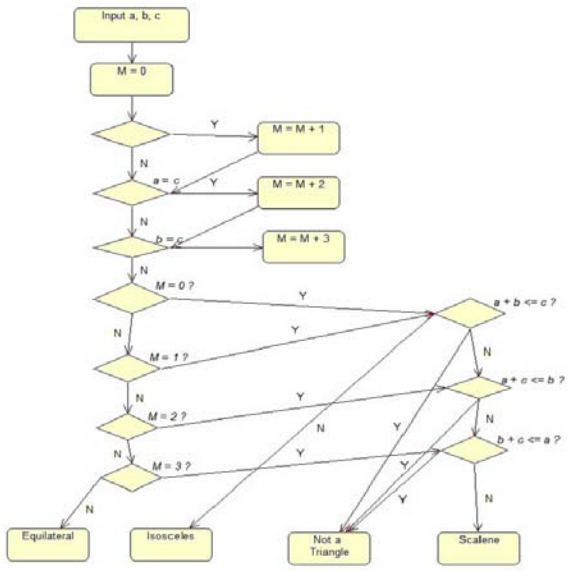


Figure 5. Program control flow graph of sample program Fig 4.

Table I
PATHS IN THE TRIANGLE PROGRAM

Program Graph Nodes	D-D Path name	Case of Definition
3	First	1
4, 5	A	5
6, 7, 26	B	5
27	C	3
28	D	4
29	E	4
30	F	3
9, 10, 31	G	5
32	H	3
33	I	3
34	J	4
35	K	3
12, 13, 36	L	5
37	M	3
38	N	3
39	O	4
40	P	3
15	Q	5
20, 21	R	5
16, 17	S	5
18, 19	T	5
22, 24, 25	U	3
22	W	2
40, 41, 42, 43	End	3

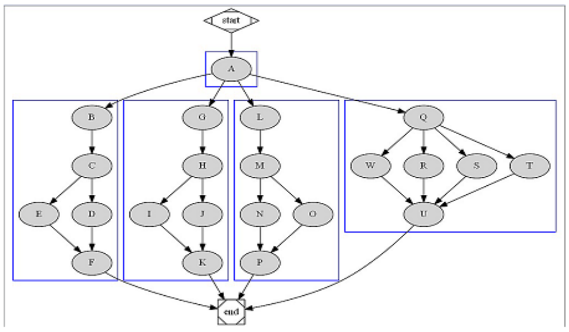


Figure 6. Generated non redundant Control Flow Graph of program.

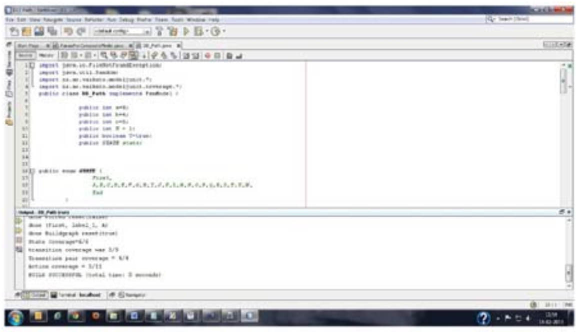


Figure 7. achieved coverage with test sequences.

The final sets of test scenarios are given in Table II

Table II
PATHS IN THE TRIANGLE PROGRAM

Path	Test Sequences	Description
p1	start-A-B-C-C-E-F-end	Isosceles
p2	start-A-B-C-D-F-end	Not a Triangle (a + c $\leq$ b)
p3	start-A-G-H-I-K-end	Not a Triangle (a + c $\leq$ b)
p4	start-A-G-H-J-K-end	Isosceles
p5	start-A-L-M-N-P-end	Not a Triangle (b + c $\leq$ a)
p6	start-A-L-M-O-P-end	Isosceles
p7	start-A-Q-W-U-end	scalene
p8	start-A-Q-R-U-end	Not a Triangle (a + c $\leq$ b)
p9	start-A-Q-S-U-end	Not a Triangle (a + b $\leq$ c)
p10	start-A-Q-T-U-end	Not a Triangle (b + c $\leq$ a)

VI. CONCLUSION

We have proposed a methodology to generate test Sequences using Path Coverage. Our technique achieves many important coverages like basis path coverage, node coverage and edge coverage. Our future work is to generate test sequences using Tabu Search algorithm and Firefly search algorithm.

REFERENCES

[1] A. Bertolino and F. Basanieri. A practical approach to UML-based derivation of integration tests. Proceedings of the 4th International Software Quality Week Europe and International Internet Quality Week Europe, 2000.

[2] J.R. Birt and R. Sitte. Optimizing testing efficiency with error-prone path identification and genetic algorithms. In Software Engineering Conference, 2004. Proceedings. 2004 Australian, pages 106 – 115, 2004.

[3] Y. Cao, C. Hu, and L. Li. Search-based multi-paths test data generation for structure-oriented testing. In Proceedings of the first ACM/SIGEVO Summit on Genetic and Evolutionary Computation, GEC ’09, pages 25–32, New York, NY, USA, 2009. ACM.

[4] J. Duran, W. Ntafos, and C. Simeon. An evaluation of random testing software engineering. IEEE Transactions, 10(4):438–444, July 1984.

[5] Gold and Robert. Control flow graphs and code coverage. Int. J. Appl.Math. Comput. Sci., 20(4):739–749, December 2010.

[6] Zhang Guangmei, Chen Rui, Li Xiaowei, and Han Congying. The automatic generation of basis set of path for path testing. In Test Symposium, 2005. Proceedings. 14th Asian, pages 46 – 51, dec. 2005.

[7] D. Lorenzoli, L. Mariani, and M. Pezze. Automatic generation of software behavioral models. In Software Engineering, 2008. ICSE ’08. ACM/IEEE 30th International Conference on, pages 501 –510, may 2008.

[8] R. Malhotra and M. Garg. An adequcy based test data generation technique using genetic algorithm. Journal of information processing systems, 7(2), June 2011.

[9] D. Qingfeng and D. Xiao. An improved algorithm for basis path testing. In Business Management and Electronic Information (BMEI), 2011 International Conference on, volume 3, pages 175 –178, may 2011.

[10] Z. Zhonglin and M. Lingxia. An improved method of acquiring basis path for software testing. In Computer Science and Education (ICCSE), 2010 5th International Conference on, pages 1891 –1894, aug. 2010.