An Improved Method of Acquiring Basis Path for Software Testing

Zhang Zhonglin, Mei Lingxia
School of electronic and information engineering
Lanzhou Jiaotong University
Lanzhou, China

***Abstract*—Basis path testing, as an important method of white box testing, uses cyclomatic complexity to define a basic set of feasible paths. But use this method to generate a set of linearly independent paths, when data dependence exists in variables involved in decision node before and after, many basis paths themselves are infeasible. This paper combines the baseline method with the dependence relationship analysis, can avoid selecting infeasible paths from the control flow graph. Example proves that it is effective.**

***Index Terms*—basis path testing, cyclomatic complexity, infeasible path**

I. INTRODUCTION

Software testing, an important way to assure software quality, is an important part of software engineering. Software testing can be divided into functional testing and structural testing from the perspective of how to select test cases. Functional testing and structural testing play different roles in software testing. Structural testing, also known as white box testing or logic-driven testing, is the process of testing the internal logic for the program. Path coverage, which widely used in unit testing, is a very important method in white box testing. Basis path testing is combination of path testing and branch testing, as path testing requires all the paths selected in computer program have to be tested, while branch testing requires possible outcomes of each branch in the program should be executed [1].

Basis path testing was proposed by McCabe in the 1980s. This method derives a set of feasible independent paths to design corresponding test cases by analyzing cyclomatic complexity of the control structure. As cyclomatic complexity that bases on control flow graph disguises some important information of code, many basis paths themselves are infeasible, that is, no test data can execute these paths.

In this paper, control flow graph is finished based on code analysis, and we particularly concern about condition predicates by using dependence analysis to make some change on control flow graph to avoid selecting infeasible basis paths.

II. BASIC CONCEPTS AND DEFINITIONS

A control flow graph (CFG) is a directed graph that consists of two types: node and control flow. (1) Node: expressed by a labeled circle, representing one or more statements, decision condition, procedures of program, or convergence of two or more nodes. (2) Control flow: expressed by arc with arrow or line, can be called an edge, representing the program control flow.

In a CFG, a node including condition is called a predicate node, and edges from the predicate node must converge at a certain node. Area defined by edges and nodes is referred to as region.

Cyclomatic complexity can be used to measure complexity of judge structure of a module. McCabe was also given calculation formula of complexity of a program structure.

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]. According to graph theory, in a strongly connected directed graph G, the cyclomatic number is defined as V (G) = m – n + p, where m is number of arcs in the graph G, n is number of nodes, and p is number of strongly connected components. For a program that has a single entry and exit point, the entire module has only one program, then p = 1. Program control flow graphs are not strongly connected, but they become strongly connected when a "virtual edge" is added connecting the exit node to the entry node [2], thus the complexity number of the module is e – n + 1. Without the "virtual edge", control flow graphs can be served as undirected graphs, thus the cyclomatic number can be calculated like this: V (G) = e – n + 2.

The second method of calculating cyclomatic complexity is that V (G) = p + 1, where p is the number of binary decision predicates. The third method is V (G) = the region number of a CFG (it refers to a strongly connected graph).

Definition 1 [3] Data Dependence: Let m, n be nodes of a CFG, v is a variable, if the three below conditions hold, then m is data dependent on n about variable v, denoted by DD (m, n, v) or DD (m, n).

- (1) n defines v, that is, $v \in def[n]$;
- (2) m uses the value of v in execution process, that is $v \in use[m]$;
- (3) There is an executable path from n to m; for any statement in this path, it does not redefine v.

Variables can be used in two ways: One is used to calculate new data or print as output results, such use is called computational use; the other is used to calculate control transfer direction of the predicate, such use is called predicate use.

Definition 2 [3]Control Dependence: Let m, n be nodes of a CFG, if the three below conditions hold, then n is control dependent on m, denoted by CD (n, m).

- (1) There is an executable path P from n to m;
- (2) For each node n'(except m, n)in P, n is a post-dominator node of it;
- (3) n is not a post-dominator node of m.

From definition 2 we know branch edges of a program can show control dependence.

III. BASIS PATH TESTING

A. *Steps of Basis Path Testing*

The basic principle of basis path testing is that all independent paths of the program have to be tested at least once. An independent path is a path which introduces a new statement or a new condition at least.

Main steps are as follows:

First, generate the CFG according to the program being test.

Second, calculate the cyclomatic complexity of the CFG based on formula, the cyclomatic complexity equal the number of linearly independent paths.

Third, generate set of basis paths based on the baseline method: 1) Select a baseline path, the "baseline" path is the first path, which is typically picked by the tester, and it should be, in the tester's judgment, the most important path to test [2]. 2) Backtrack the baseline path by flipping each decision node to generate new paths. The basic set of paths is different while choosing different paths.

Finally, eliminate infeasible paths; add other important paths; and generate test cases to make each path can be executed.

B. *Example Analysis*

As an example, consider the following program, which used to determine shape of a triangle. The CFG is shown in Fig. 1, where node 4, node 11, node 12 and node 14 are all represent ends of the condition statements that are "endif".

```
void Triangle ( int a, int b, int c )
{
    bool  isTriangle;
1    if ( (a<b+c) && (b<a+c) && (c<a+b) )
2        isTriangle=true;
    else
3        isTriangle=false;
5   if ( isTriangle )
    {
6       if ( a==b && b==c )
7          printf ("Equilateral\n");
        else
8          if (a!=b && b!=c && a!=c)
9             printf ("Scalene\n");

          else
10           printf ("Isosceles\n");
    }
    else
13      printf ("Not a Triangle\n");
}
```

The analysis process of basis path testing is shown as follows:

1) From Fig. 1 we can calculate the complexity V (G) = e – n + 2 = 17 – 14 + 2 = 5; or V (G) = p + 1 = 4 + 1 = 5.

2) Select the baseline path

P1:1→3→4→5→13→14.

3) On the basis of the baseline path, flip each decision node to get the other four paths, they are as follows:

P2:1→2→4→5→13→14 (node 1 is flipped);

P3:1→3→4→5→6→7→12→14 (node 5 is flipped);

P4:1→3→4→5→6→8→9→11→12→14 (node 6 is flipped);

P5:1→3→4→5→6→8→10→11→12→14 (node 8 is flipped).

IV. AVOID SELECTING INFEASIBLE PATHS

Basis path testing is based on the CFG, it is away from the source code, therefore the paths are logistically feasible, but in fact some of the paths can not be executed at all. Infeasible path mainly occurred on the condition that decision nodes are series connection, and variables involved in decision node before and after have a certain degree of dependency.

Checking the program code of the above basis paths, we can see that P2, P3, P4 and P5 are infeasible paths. This is because the first decision node assigns a value to variable "isTriangle" and node 5 uses the value as a condition decision; it leads close correlation between the two decision nodes. The outcome of the first decision determines the outcome of the second, that is, when variable "isTriangle" is given "true" value at statement 2, execution should transfer directly to node 6 after

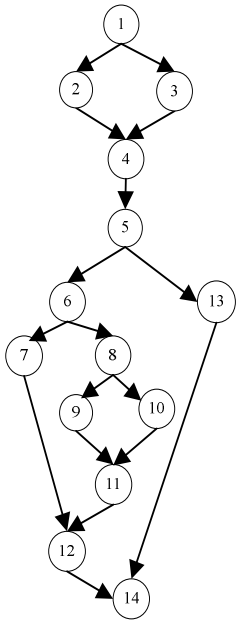


Figure 1. The CFG of the function "Triangle".

node 5 is executed. Similarly, when "isTriangle" is given "false" value at statement 3, execution should proceed to node 13 after node 5 is executed.

In order to avoid selecting infeasible basis paths, we analyze the source code before generating the CFG. We take great emphasis on branch predicates, the process is shown as follows:

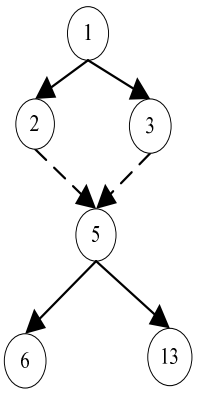
1) Extract the predicate that reference non-initial input variable; analyze the data and direct control dependences of the predicate.

In the above function "Triangle", predicate nodes are node 1, node 5, node 6, node 8, and the variable "Triangle", which node 5 referenced, is not an initial input, its data and direct control dependences are shown in Fig. 2, where solid line represents control dependence and dotted line represents data dependence.

2) In the CFG, the node that has double data dependence is marked in bold, meaning this node will lead the branch execute along a define orientation. Considering nodes have dependence relationship with the bold node, edges from nodes that the bold node are data dependent on to the bold node are marked respectively, the edges can be distinguished by solid line and dotted line. As data dependence nodes have influence on nodes that are direct control dependent on bold node, thus edges from the bold node to its directly control dependence nodes which corresponding to data dependence nodes are also marked, respectively.

In the example, node 5 is shown in bold, the edge from node 3 to node 5 is marked by a dotted line, though the edge from node 2 to node 5 is a solid line, because node 3 will lead execution transfer from node 5 to node 13, therefore the edge from node 5 to node 13 is also represented with a dotted line. It is shown in Fig. 3.

3) When selecting paths, if we meet the bold node we can not flip it. If an edge points to a bold node is a solid line, we select the node as a next node whose edge is also a solid line from the bold node. If an edge points to a bold node is a dotted line, we select the node as a next node whose edge is also a dotted line from the bold node. For the non-bold marked predicate node, we can select a branch of it arbitrary, and the node can be flipped to get a new path.



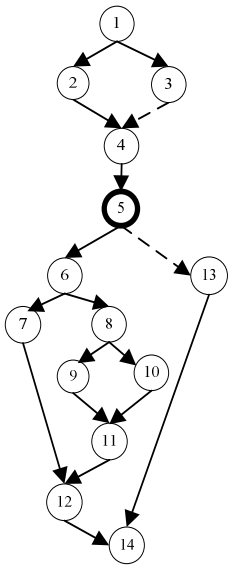


Figure 2. Dependences of node 5 Figure 3. The modified CFG of the function "Triangle"

We should cautious to pick the baseline path. Usually, requirements of selecting the baseline path are [4, 5] a) the path should be as long as possible; b) the path as much as possible through branch nodes. However, in the tester's judgment, the shortest path should be the most important path of the system [6]. It is easier for the shortest path than long path to judge whether the function it represents is legitimate. So we can select the shortest path that represents a legitimate function to be the baseline path.

In Fig. 3, path: 1→3→4→5→13→14 and path: 1→2→4→5→13→14 are the shortest paths, but node 5 is a bold node, the edge from node 3 to node 5 is a dotted line (node 4 is the end node for the statement, it does not affect our selecting), the edge from node 5 to node 13 is also a dotted line, node 5 should be selected after node 3, the path we get is

1→3→4→5→13→14, it is the same like P1. And the path is feasible, so we select P1 to be the baseline path.

On the basis of the baseline path P1, we can select other paths:
P2':1→2→4→5→6→7→12→14 (select the solid line after node 5 is executed);
P3':1→2→4→5→6→8→9→11→12→14 (node 6 is flipped);
P4':1→2→4→5→6→8→10→11→12→14 (node 8 is flipped).

We can judge that these four paths are all feasible, and then we can design test cases to generate test data. However, the path number of basic set is 4, less than 5, the cyclomatic complexity. This is because the control flow graph does not contain processing information of a program; the logical relationship among nodes can not be shown clearly, therefore, in basis path testing, the cyclomatic complexity should be the upper limit for the number of basic paths [7].

V. CONCLUSION

Basis path testing is a widely used method of white box testing. The method uses cyclomatic complexity as the basis for finding a set of independent paths to express the other paths. This paper focuses on the problem that variables involved in decision node before and after have data dependence. We analyze dependence relationship of the predicate; pick the shortest path as the baseline path instead of the longest path which has much branch nodes. In this way, using "baseline path + flip" to generate set of independent paths, may avoid selecting infeasible paths. Of course, in the process of programming, avoid data dependence is a better way to simplify the testing process, and to improve software quality.

REFERENCES

[1] K. Mustafa and R. A. Khan, Software Testing: Concepts and Practices, Beijing: Science Press, 2009.

[2] Arthur H. Watson and Thomas J. McCabe, "Structured testing: a testing methodology using the cyclomatic complexity metric," NIST Special Publication, September 1996.

[3] Li Bixin, Program Slicing Technique and Its Applications, Beijing: Science Press, 2006.

[4] Wu Jianjie, Chen Chuanbo, and Xiao Laiyuan, Basis of Software Testing Technique, Wuhan: HuaZhong University of Science and Technology Press, 2008.

[5] Du Qingfeng, Li Na. “White box test basic path algorithm,” Computer Engineering, vol. 35, pp. 100–102,123, 2009, 08.

[6] Marnie L.Hutcheson, Software Testing Fundamentals Methods and Metrics, Posts&Telecom Press, 2007, 09.

[7] Sun Haiying, Method and Practice of Software Testing, Beijing: China Railway Publishing House, 2009.