An Improved Algorithm for Basis Path Testing

Du Qingfeng, Dong Xiao
School of Software Engineering
Tongji University
Shanghai, China
du_cloud@sohu.com

***Abstract*—Basis path testing is an important test method in white box testing. This method generates a set of linearly independent paths, which we called basis paths, from Control Flow Graph (CFG) and all the other paths can be expressed by them. However, when applying these basis paths to designing test cases, we will always find that some of them are infeasible when understanding from business logic. In this paper we put forward a new idea to simply CFG, at the same time to avoid the generation of infeasible paths in the set of basis paths.**

Keywords-white box testing; control flow graph; basis path testing; method improvement

I. INTRODUCTION

With the development of the software industry, software testing gradually takes a more important role in order to assure the software quality. White box testing, which is also called structural testing or logic-driven testing, mainly focus on the internal logic test of the code. Its objective is to achieve specific logical coverage indicator, such as statement coverage, condition coverage, branch coverage, basis path coverage and so on. Test cases are always designed according to the Control Flow Graphs (CFGs).

Basis path testing was proposed by Thomas McCabe in the eighties of last century. Based on cyclomatic complexity measure to CFG and specific algorithm, independent basis paths can be created and test cases can be design according to these paths.

McCabe considered that the set of all possible paths in the program can be taken as a vector space, and then there must be a group of basis can cover the entire vector space by linear combination. However, McCabe's basis path approach is purely founded on the Mathematical vector theory, in practice, some basis paths are infeasible in business logic, and they'll become meaningless when we design test cases, making the test work more difficult.

In this paper, we try to focus on the simplification of the CFG in order to avoid the generation of infeasible test paths. It will be proved workable and effective by the examples.

II. CONCEPTS

A. Vector Space and Basis

Vector space is also called linear space. Let V be the set of n-dimensional vector, if set V is non-empty, and closed under finite vector addition and scalar multiplication, where the scalars are members of a field F, we say V is a vector space over F. furthermore, a vector space should satisfy some axioms, we won't list all that here and details could refer to Linear Algebra.

If a vector set $\{v_1, v_2, \dots, v_k\}$ in a vector space V is linear independent and this set spans V, we say that elements $v_1, v_2, \dots, v_k$ form a basis of V, which means that all v_n in V can be expressed uniquely as a linear combination.

B. Control Flow Graph

CFG describes the logic structure of software components. Each CFG consists of nodes and edges. The nodes represent computational statements or expressions, and the edges represent transfer of control between nodes. Each possible execution path of a software module has a corresponding path from the entry to the exit node of the module's control flow graph. This correspondence is the foundation for the structured testing methodology [4].

C. Cyclomatic Complexity

Cyclomatic complexity is also defined as V (G) while v means the cyclomatic number in graph theory and G stands for that the complexity is a function of the graph. It is entirely based on the structure of CFG, and measures the amount of decision logic in a single software module [4]. For each module, we have many formulas to calculate the cyclomatic complexity and the most common one is $V(G) = e - n + 2$, where e represents the number of edges in the CFG and n for nodes.

Combined with the concepts of vector space and basis we mentioned before, we can give that cyclomatic complexity is precisely the minimum number of paths that can, in (linear) combination, generate all possible paths through the module [4]. And corresponding to the linear alphabet theory, these basis paths are just like the basis of a vector space.

III. THE BASELINE METHOD

The baseline method is different from the basis path testing. The former is a technique to derive a set of basis paths through the CFG generated from the tested component, and it is equal to the cyclomatic complexity in number. The idea is to start with a baseline path, then vary exactly one decision outcome to generate each successive path until all decision outcomes have been varied, at which time a basis will have been generated [4]. Suppose that we have got a CFG from the program to be tested, with the baseline method to do the basis path testing, we can

978-1-61284-109-0/11/$26.00 ©2011 IEEE

generally follow the next steps:

Step 1 Pick a functional baseline path, which is the most important path to test in the tester's judgment. In most cases, the baseline path we chose should satisfy the typical business logic and try to pass more decision nodes.

Step 2 To generate the next path, change the outcome of the first decision along the baseline path while keeping the maximum number of other decision outcomes the same as the baseline path. And begin again with the baseline but vary the second decision outcome rather than the first to generate the third path. Repeat the flip operation until all decision nodes has been done, and the basis path set is complete.

Sometimes we can use some tools to make sure that the set of paths we got is actually linear independent. Then we design test cases based on this set and add some exceptional conditions test cases if necessary.

IV. Improved Basis Path Testing Method

Although the baseline method is mathematically rigorous, the problem is that often there are some paths are infeasible actually, which means when generating test cases, these paths cannot be executed at all. In most CFGs, 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 [1]. In other words, if an "end-judge" node is not the final exit of a CFG and it is followed by another "if" or "switch-case" statement, there could be infeasible path occurred.

Normally, we have two ways to deal with the infeasible paths:

- Find out the out-degree which is greater than or equal to two node in the infeasible path and flip the following branch until getting a feasible path;
- Modify the infeasible paths manually to meet the business logic.

These two methods are both based on the existing CFG and may require a lot of manual attempts. But actually, we want the changes in human intervention could be as few as possible when generating the basis paths. For this idea, we decided to start from the CFGs.

For our improved method, suppose one code fragment including many parts of judgments, we should observe the following principles:

Principle 1 If the results of the previous judge determine the premise of the next judgment, the causal paths should be joined when generating the CFG.

In details, if one node forms many branches which are all depended on the results of the last judgment, we can just omit the end node of the former judgment part and the next determination node, then append the following branches to the corresponding precondition node of itself.

Principle 2 The final "end-judge" node is the exit of a CFG. This node cannot be omitted.

Principle 3 If there are compound conditional statements, the CFG should be able to display the right condition of splitting the compound conditional statement into separate ones.

Here we give a familiar example – the triangle shape judgment program - to do some further analysis. The program fragment and the corresponding CFG are shown as below.

```
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              print ("Equilateral\n");
          else
8              if (a != b && b != c && a != c)
9                   print ("Scalene\n");
               else
10                  print ("Isosceles\n");
      }
      else
13         print ("Not a triangle\n");
}
```

With the traditional baseline method, we first calculate the cyclomatic complexityV (G) = e – n + 2 = 17 -14 + 2 = 5 from the CFG, then get the five basis paths as follows:

$\mathbf{P_1}$: 1→2→4→5→6→8→9→11→12→14 (baseline);

$\mathbf{P_2}$: 1→3→4→5→6→8→9→11→12→14 (node 1 flipped);

$\mathbf{P_3}$: 1→2→4→5→13→14 (node 5 flipped);

$\mathbf{P_4}$: 1→2→4→5→6→7→12→14 (node 6 flipped);

$\mathbf{P_5}$: 1→2→4→5→6→8→10→11→12→14 (node 8 flipped).

We can easily find that $\mathbf{P_2}$ and $\mathbf{P_3}$ are invalid for test. For these two paths, since we have determined whether it is a triangle from node 1 to 4, the following path from node 5 is completely contradictory from the judge before.

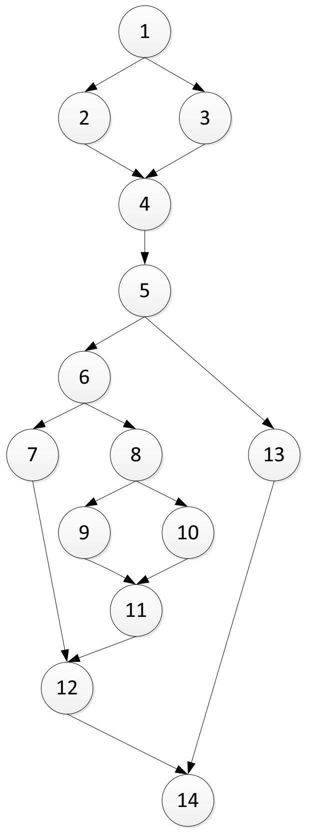


Figure 1. The Control Flow Graph from the function Triangle

To avoid the infeasible paths by the improved basis path testing method, let's focus on the Triangle code fragment first, it can be divided into two parts: one part starts from the beginning to line 3 determines whether it is a triangle; and the second from line 5 to the end determines the shape of a triangle or prints out the non-triangle information. For the executable test paths, the results of the first part decide the branch choice of the second one.

After clearing the role of each part, let's pay attention to the CFG generated. We can see that node 4, 11, 12 and 14 all represent the end of a condition statement. Suppose we can allow the follow-up operations corresponds to the determine result of the previous part, these "end-judge" nodes should be omitted, thus we achieve the simplification of the CFG.

Follow this idea, only in the determination of the premise of the triangle needs to judge the shape and non-triangle just has to print a message. In this correspondence, the CFG in Fig. 1 could be modified to Fig. 2.

In this CFG we merge the two "if" fragments together so that after the first judgment of node 1, we can ensure every path based on a reasonable premise. In this condition, the cyclomatic complexity and the number of basis path is reduced to 4 and all paths are feasible. Applying the baseline method again we can get these 4 paths as below:

P$_1$: 1→2→6→8→9→11→12→14 (baseline);

P$_2$: 1→3→13→14 (node 1 flipped);

P$_3$: 1→2→6→7→12→14 (node 6 flipped);

P$_4$: 1→2→6→8→10→11→12→14 (node 8 flipped).

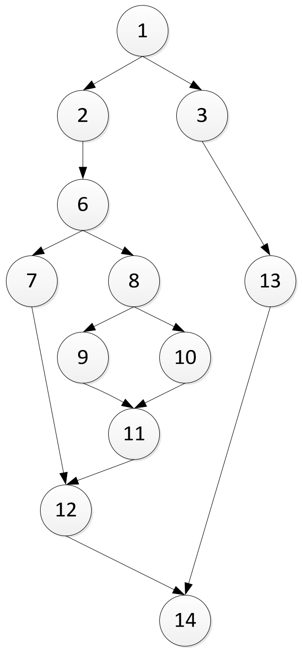


Figure 2. The modified Control Flow Graph for function Triangle

According to these basis paths we can design the reasonable regular test cases and the corresponding results:

TC$_1$: Is a triangle and is Scalene;

TC$_2$: Is not a triangle;

TC$_3$: Is a triangle and is Equilateral;

TC$_4$: Is a triangle and is Isosceles.

Furthermore, for the compound conditional statements can be split into smaller single conditional statement, let's take the line 1 in the program as an example. We can get a part of the CFG as Fig. 3:

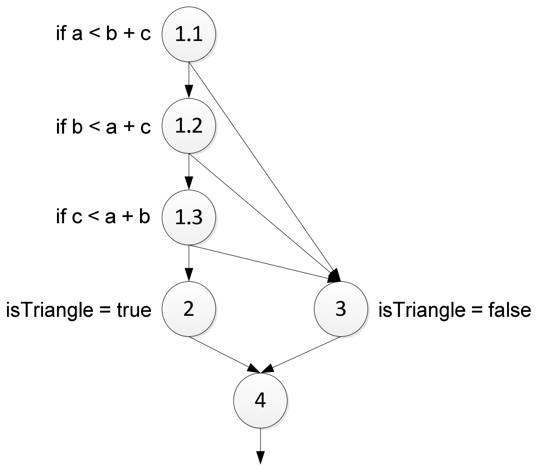
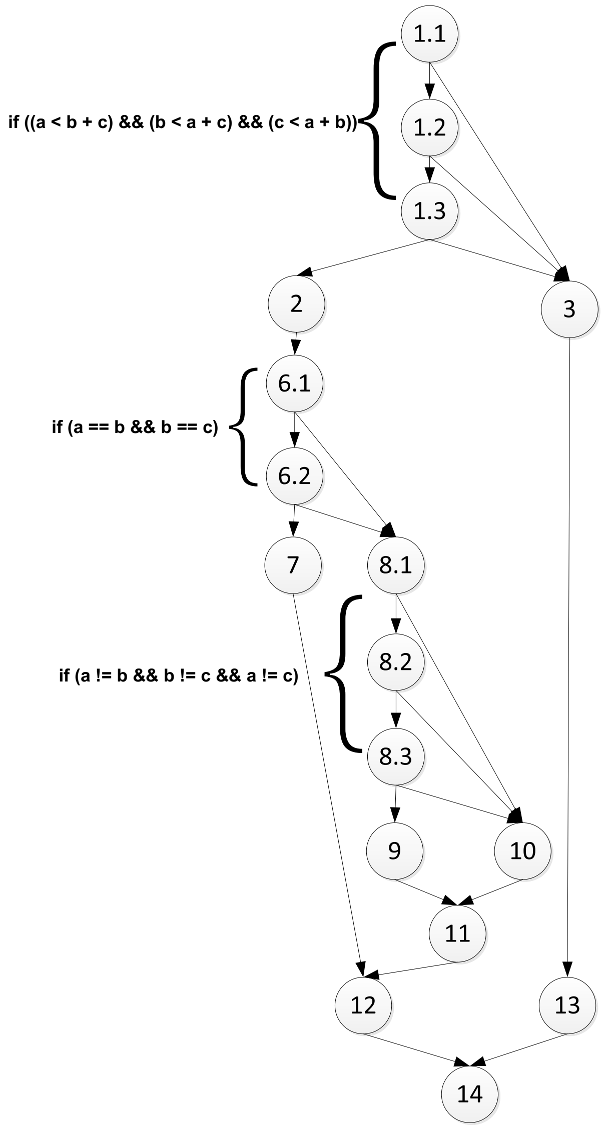


Figure 3. Part of the split Control Flow Graph for function Triangle

In this case, our method is still workable, and we can get a complete CFG modified as Fig. 4:



if ((a < b + c) && (b < a + c) && (c < a + b))
1.1
1.2
1.3
2
3
if (a == b && b == c)
6.1
6.2
7
8.1
if (a != b && b != c && a != c)
8.2
8.3
9
10
11
12
13
14

Figure 4. The split and modified Control Flow Graph for function Triangle

Although the number of basis paths increased, we can insure that every path is feasible.

For this Triangle example we do one note at last. In our modified CFGs, node 11, 12, 13 and 14 are all "end" node, in fact we can merge them together to get a simpler CFG, and other situations will not be changed.

At the end of this section we finally make some additional explanation. Our improved basis path testing method can be applied not only for the program including "if" statement for judge, but for the "switch-case" statement as well. We will not go into details here.

V. Conclusion

Basis path testing is an important and wildly applied white box testing technology. It is based on the cyclomatic complexity theory by McCabe and always generates the basis path set through the baseline method. For it fully depends on the CFG of the program, some basis paths we get are feasible in mathematics but inexecutable in the actual test. To avoid this condition, we give an improved method that when generating the CFG, connect the causal paths of the two series judgment parts and omit the intermediate nodes. In this ideology, we construct a correspondence between the premise and the following branch, to make sure that every basis path is feasible. Under the condition that the results of the previous judge fragment are the premise of the next judgment parts, this method makes sense.

References

[1] Zhang Zhonglin and Mei Lingxia, "An Improved Method of Acquiring Basis Path for Software Testing," 5th International Conference on Computer Science and Education, ICCSE 2010, pp.1891-1894, 2010..

[2] Du Qingfeng and Li Na, "White box test basic path algorithm," Computer Engineering, vol. 35, pp. 100–102,123, Augest 2009.

[3] Andreas Spillner, Tilo Linz, and Hans Schaefer, "Software Testing Foundations: A Study Guide for the Certified Tester Exam," O'Reilly Media, 2006.

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