The splitting and matching algorithm of dynamic path oriented the function calling relationship

Yongmin Mu[1], Huili Li[2], Bing Jiang[3], Xuefei Liu[4], Miao Wu[5]
Open Computer System Laboratory
Beijing Information Science & Technology University
yongminmu@163.com, Leeyanlichina@126.com, zt0414324@yahoo.com.cn, xuefeiliu@hotmail.com, 1048616266@qq.com

***Abstract*—For the huge amount of the logical path, the path coverage technology of statement level can't be realized. On the premise that each function has finished the unit testing, the path coverage algorithm based on the function calling relationship can guarantee the adequacy of the test. Get the dynamic path by running the instrumented source code, then split the dynamic path using the splitting algorithm. Match the split dynamic path called subset of the global static path set with the global static path set using the matching algorithm. The coverage rate can be figured out with the matching result. The experiment results show the splitting algorithm and the matching algorithm can obtain the coverage rate accurately.**

Keywords-function calling relationship; dynamic path ; static path; coverage rate

I. INTRODUCTION

Software testing, which is the key to guarantee the software quality and improve the software reliability, is an important stage in the software development process [1]. White-box testing mainly tests the logical paths of the source code and requires covering the structural features of the program being tested to some extent, and then it can evaluate the adequacy of the testing based on the criterion that whether some kinds of the software components can be tested adequately. Therefore, white-box testing can be called a testing technology based on the coverage sometimes[2]. For example, statement coverage is a logical coverage criterion, which requires all the program statements to be run .

Up to now, there are many coverage technologies in the coverage analysis of software testing. In these coverage technologies, the path coverage technology has the highest coverage rate, which requires each possible path in the program to be covered at least one time. But the actual situation is that even the program of very small scale contains so many logical paths. So the path test method is still remaining on a theoretical level [3]. In a large program, if there are too many branches and loops , the number of paths will increase rapidly and the complete path coverage testing is infeasible. Generally, the tests only choose a finite subset of all the complete logical paths according to the related standards [4]. For example, the basic path coverage testing chooses the subset of paths which can cover all the states in the system [5].

On the premise that each function has finished the unit testing, this paper uses another testing coverage criterion which is called "the path coverage criterion based on the function calling relationship". This criterion extends the particle size of code analysis from the statement to the function. It can avoid the explosive growth of the paths and guarantee the test security to some extent [6]. The method proposed in this paper can increase the practicality of the tests.

II. BASIC CONCEPTS

A. Function Calling Relationship

The function calling relationship, whose basic unit is function, can be obtained by analyzing the control logical relationship among the functions in the source code. It is different from the function including relationship which only analyzes what functions a function in the source code includes without considering the logical relationship among the functions.

B. The Global Static Path Set

The static path is the function sequences based on the function calling relationship from entry point to the exit point of the program using static analysis method [7]. The global static path set is the set of all the static paths. Expressed as $B(S,C)=\{P_1, P_2,\ldots, P_n\}$, the symbol S represents the source code, the symbol C represents the rules of the function calling relationship, the symbol P_i represents one static path. For example, the source code is as follows:

```
a(){
a();}
main(){
a();
while(cond){
   if(cond){
     b();}
else
     c();}
d();}
```

In the source code, there are four functions which are main, a, b and c. And according to the function calling relationship, there are three static paths: main->a->a->b->d, main->a->a->c->d, main->a->a->d. And the global static path set is S={main->a->a->b->d; main->a->a->c->d, main->a->a->d}.

C. Dynamic path

Dynamic path is the sequence of the value of the pile points obtained by inputting the test data, compiling and

978-0-7695-5011-4/13 $26.00 © 2013 IEEE
DOI 10.1109/IHMSC.2013.229

running the instrumented source code. The dynamic path expressed as $L(S,D)=\left\{\left\langle f_{10},f_{11},c_{10}f_{20},f_{21},f_{30},\cdots c_{i1},\cdots f_{n1}\right\rangle \middle| Split\left(\left\langle f_{10},f_{11},c_{10}f_{20},f_{21},f_{30},\cdots c_{i1},\cdots f_{n1}\right\rangle\right)\in B\right\}$, the symbol S represents the instrumented source code, the symbol D represents the test data, The symbol f_{i0} represents the value of the start pile point of the function. The symbol f_{i1} represents the value of the end pile point of the function. The symbol c_{i0} represents the value of the true pile point of the control logical keywords. The symbol c_{i1} represents the value of the false pile point of the control logical keywords. The symbol $Split(\ )$ represents the splitting algorithm. And the symbol B represents the static path set of the source code.

III. THE SPLITTING AND MATCHING ALGORITHMS OF DYNAMIC PATH

Splitting the dynamic path requires knowing the features of the static path and dynamic path.

A. The features of the static path

Taking the program above for example, all the static paths have the following features [8]:

- If there are recursions, the static path only contains a layer of the recursive function calling. For example, the recursive function names “a” can be expressed as a->a;
- If there are loops, the static paths only contain all the cases of the first layer circle. In the above code the static paths of the first layer circle are {a->b; a->c}.

B. The features of the dynamic path

Because the dynamic path may contain many recursive function calling and multilayer cycles, the algorithm splits the dynamic path into paths which only contain one layer of recursion and circle according to the recursion and loop, in order to match the static paths.

Taking the program source code for example, the execution path of the program probably are main->a->a->a->b->b->b->c->c->c->b->d. According to the features of the static path, the path can be split into two static paths :{ main->a->a->b->d; main->a->c->d}.

C. Obtaining the dynamic path

In order to obtain the feature of the dynamic path of the program, the instrumentation of the source code is necessary. Considering the features of static path and dynamic path, we need to obtain the execution sequence of the function names and the control logical keywords. So the pile points should be located in the entry of the functions , the exit of the functions and the keywords position.

The size of the pile function can influence the measured system directly. The smaller the pile function is and the less the pile points are, the less influence does the instrumentation have on the measured system and the lower the code inflation rate is. So we propose a design scheme of the pile functions, which is only inserting a hexadecimal marker into the positions which need to be marked.

The instrumentation information can be recorded by files because the amount of the information collected by instrumentation is not large. The recorded information contains two kinds. One is a list of the function objects which contains the function names, the entry points of the functions(STag) and the export point of the functions(ETag). The other is a list of the branch objects which contains the keywords, names of the functions which contains certain keywords, the true pile points of the keywords(TTag) and the false pile points of the keywords(FTag).

D. Splitting algorithm

In order to take the function path coverage tests, we need to design relevant test cases. Then we can obtain the dynamic path when the program runs according to the test cases. The splitting algorithm splits the dynamic paths into subset of the global static path set according to the function calling relationship and realizes the corresponding relation between the test cases and the test requirements. Considering the features of the static path and dynamic path, we design the process of the splitting algorithm as follows:

- Insert piles into the program according to the selected pile points and the well-designed instrumentation functions.
- Run the instrumented source code and obtain the information data of the instrumentation records. Then record the data by files.
- Create array staticPath to store the splitting results. Then save the list of function objects into the array named getFuncObject. Save the list of branch objects into the array named getBranchObject. Save the list of dynamic path into the array named dataList.
- Fetch out the values of the pile points one by one from the dataList. And match them with the values in getFuncObject and getBranchObject. If the match with the values in getFuncObject successes, fetch out the function names directly. If the match with the values in getBranchObject successes, split the dynamic path according to the branch names.

The concrete splitting algorithm is as below:

Input：getFuncObject: Information of the function list
getBranchObject: Information of the branch list
dataList: Dynamic path

Output：staticPath：the splitting results

```
(1) begin
(2) Stack tagStack;
(3) foreach(tag in dataList)
(4) {if(tag is STag)
(5) {if(!tagStack contains tag)
(6) {push tag into tagStack;
(7) funcName=GetFuncNameByTag(tag);
                //get the function name by tag from
                getFuncObject
(8) add funcName into staticPath; }
(9) else{funcName=ProcessRecursion(tag);
```

(10) add funcName into staticPath; }}
(11) elseif(tag is TTag)
(12) {funcName=ProcessLoop(tag);
(13) add funcName into staticPath;}
(14) elseif(tag is ETag)
(15) {pop tag from tagStack;}}
(16) end

E. Matching algorithm

Match the subset of the global static path set obtained by splitting the dynamic path with the global static path obtained by analyzing the source code statically. Figure up the coverage rate according to the matching result.

To realize the match, we need to compare the global static path set with the subset of the global static path set obtained by splitting the dynamic path. There is a common comparative method which is putting the two sets in the arrays respectively and then comparing every element in the two arrays iteratively to verify that the subset of the global static path set obtained by splitting the dynamic path can match with the elements in the global static path set. Obviously, the efficiency of this method is not high. The one by one match of the elements in the array actually is the process of finding the elements in the array. Compared with the inefficient match method, the tree structure can provide the more efficient match method. So we construct the tree structure to present the global static path set and traverse the tree to look up the elements.

The concrete matching algorithm is as follows：

Input: B(S, V): The global static path set
L(S, D): The subset of the global static path set
Output: the matching result

(1) begin
(2) Tree Btree;
(3) add node Main into Btree;
(4) foreach(path in B(S,V))
(5) {currentNode=Main;
(6) foreach(node in path)
(7) {if (node in currentNode.children)
(8) {currentNode=currentNode.child;}
(9) else
(10) {add node into currentNode.childern;
(11) currentNode = node;}}}
(12) foreach(path in L(S,D))
(13) {find path in Btree;
(14) figure the path in Btree;
(15) return true;}
(16) end

IV. EXAMPLES

Fig. 1 shows an example about the instrumented source code. The instrumentation mainly happens when function names or control logical keywords appear. For a function it requires inserting start pile function CProbeS(tag) at the beginning of the function and end pile function CProbeE(tag) at the end of the function. And for the control logical keywords, insert the tag (CProbeT(tag),1) if it is true and insert the tag (CProbeF(tag),0) if it is false.

```
int f1(int i)
{CProbeS(0x75E00001);{i++;
      {CProbeE(0x563FFFFF);return i;}
}}
int f2(int i)
{CProbeS(0x74E00002);{i = i+2;
      {CProbeE(0x563FFFFE);return i;}
}}
int f3(int i)
{CProbeS(0x74A00003);{i = i+3;
      {CProbeE(0x563FFFFD);return i;}
}}
int main()
{CProbeS(0x75A00004);{scanf(" %d",&i); int i;
   while(((i<10)?(CProbeT(0x577FFFF8),1):(CProbeF(0x577FFFF7),0)))
  {
       if(((i<3)?(CProbeT(0x575FFFFA),1):(CProbeF(0x575FFFF9),0)))
            i = f1(i);
       else
            if(((i>=3&&i<6)?(CProbeT(0x575FFFFC),1):(CProbeF(0x575FFFFB),0)))
                 i = f2(i);
            else
                 i=f3(i);
  }
  {CProbeE(0x563FFFF6);return 0;}
}}
```

Figure 1. The instrumented source code

There are four static paths in the global static path set in the program, which are {main->f1; main->f2; main->f3; main->}

When the input is i=2, run the source codes being instrumented. Then we can obtain a dynamic path as Fig. 2 shows:

```
00000004->000ffff8->000ffffa->00000001->000fffff->
000ffff8->000ffff9->000ffffc->00000002->000ffffe->
000ffff8->000ffff9->000ffffc->00000002->000ffffe->
000ffff8->000ffff9->000ffffb->00000003->000ffffd->
000ffff7->000ffff6->
```

Figure 2. The dynamic path

After splitting the dynamic path and matching the global static path set, we can obtain the result as Fig. 3 shows. The test case covers three static paths in the global static path set.

```
The number of the selected test case
files is:1

C:\test programs\the dynamic paths of the
while loop\the dynamic paths of the
while loop\flow1.data

The number of the static path covered by
the dynamic is 3
1:main->f1->
2:main->f2->
3:main->f3->
```

Figure 3. The matching result

Input several groups of test data will obtain many dynamic paths. Splitting those dynamic paths can obtain an subset of the global static path, which may contains some repetitive and redundant paths. So we optimize the subset, remove the repetitive and redundant paths, and construct a new set names optimized dynamic path set. Take the data in table 1 as the experimental data , the change of the results after optimizing is as the Fig. 4 shows.

TABLE I. TESTED PROGRAM

Name of the tested program	Lines of the tested program	Number of the dynamic paths after splitting
T1	102	7
T2	216	23

T3	346	34
T4	365	41
T5	433	48
T6	436	70
T7	476	99
T8	566	146
T9	521	180

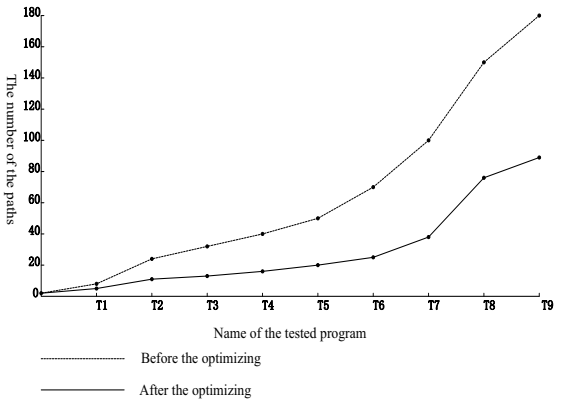

Figure 4. The change of the results after the optimizing

V. CONCLUSION

The path coverage is a severe criterion in the code coverage method. But the number of the paths in a complex program is so large that it is unavailable to cover the paths completely, the automated path testing is very difficult [9]. So this paper proposes a coverage generating technology based on the function calling relationship, which increases the feasibility of automated testing.

This paper uses the instrumentation technology and inserts piles into the source code. Then compile and run the instrumented source code by inputting test cases to obtain the dynamic paths which are corresponding with the test cases. Then analyze the dynamic path, split the recursion and circulation to obtain the subset of the global static path set which can match with the global static path set. Analyze the coverage of the function calling path according to the matching result of the global static path subsets and the global static path set, conduct the testing engineers to design test cases for the paths which are not covered.

With the increase of the software scale, the software test becomes more and more important. The design of the test cases become an important work and the automatic generating of the path test data is the key issues in the next step.

ACKNOWLEDGMENT

This work is supported by the Key Discipline Construction Foundation of Beijing (NO.PXM2013_014224_000041) , Science and Technology Funding Project of Beijing Education Committee (NO.KM201110772016) and the Key Discipline Construction Foundation of Computer Applications Technology (PXM2013_014224_000025).

REFERENCES

[1] Lv Jinhe. The Path Coverage Test. Computer & Information Technology, 2010, z1: 71-75

[2] MU Yongmin, WANG Rui, ZHANG Zhihua, and DING Yuan. Automatic Test Method Research on the Word Part of Document Format Translator[J]. CHINESE JOURNAL OF CLECTRONICS Vol.22 No.1 January 2013 P55-60

[3] W. E. Howden. Reliability of the Path Analysis Testing Strategy, IEEE Transactions on Software Testing, 1976: 208-215

[4] Zhao Liang, Wang Jianmin, Sun Jiaguang. A Study of Software Test Criterion Effectiveness Measure. Journal of Computer Research and Development. 2006, 43(8): 1457-1463

[5] Qiu Xiaokang, Li Xuandong. A Path-Oriented Tool Supporting for Testing. Acta Electronica Sinica, 2004, 32(12): 231-234

[6] Zhang Zhihua, Mu Yongmin. Research of Path Coverage Generation Techniques Based Function Call Graph. Acta Electronica Sinica, 2010, 38(8) : 1808-1811

[7] Bing JIANG, Yongmin MU, Zhihua ZHANG. Research of Optimization Algorithm for Path-Based Regression Testing Suit. The Second International Workshop on Education Technology and Computer Science, ETCS2010. WuHan: IEEE Computer Society, 2010. 303-306

[8] Huiyu ZHENG, Yongmin MU, Zhihua Zhang. Research on the Static Function Call Path Generat- ing Automatically. 2010 The Second IEEE International Conference on Information Management and Engineering. ChengDu: 2010. 405-409

[9] MU Yongmin, ZHENG Yuhui, ZHANG Zhihua, LIU Mengting.The algorithm of infeasible paths extraction oriented the function calling relationship. CHINESE JOURNAL OF ELECTRONICS Vol.21 No.2 April 2012 ISSN 1022-4653.