International Journal of Hybrid Information Technology
Vol. 9, No.7 (2016), pp. 429-442
http://dx.doi.org/10.14257/ijhit.2016.9.7.39

Research for Java Path Test Tracking Method Based on FCP

Chen Liu, Mu Yong-Min and Yan Ming-Ming

Beijing Information Science and Technology University, Beijing 100101, China

Abstract

The software system is made up of many function calling paths (FCP), every function calling path is a complete business process. Due to the huge FCP, manual testing is time-consuming and test paths are easy to be confused. The software testing process tracking can effectively remove the redundant test cases, find tested and untested paths, and mark the test cases. This paper combines the technique of Soot thought to analyze Java codes, extracts function calling relationships, and generates FCP. The dynamic paths of Java function are obtained by instrumentation technology. Finally, the dynamic path tracking algorithm is used to determine the range of tested paths. The experimental result shows that, the algorithm proposed in this paper is effective for tracking the Java function calling paths in the testing process, and put forward a new idea for the study of other language testing process tracking.

Keywords*: software testing; Java; function calling paths; the testing process tracking; Soot; instrumentation*

1. Introduction

Program tracking technology first appeared in 1988, the representative research results is Bugnet, it is a kind of tool platform for debugging distributed parallel programs written in C language on the UNIX system. The research of software defect tracking and locating began in the 1970s. At the earliest, Weiser proposed program slicing can be used for program comprehension and software debugging. Foreign experts have made significant contributions in the dynamic tracking. In 1981 Neivergeit and Plattner published papers: Monitoring Program Execution: A survey[23]. This paper made a comprehensive analysis and study on dynamic tracking technology, and it discussed several basic concepts, structure and target *etc*. Since then, researchers used dynamic monitoring technology to each field, obtained a series of achievements, especially had a very good application in Java language. At present, there are many dynamic tracking and monitoring system based on Java. Mainly the following: Java-MOP[24,25], IncETCJ[26], JPaX[27,28], Java-MaC[29,30].

Now the domestic research results are also very rich. However, no study on the testing process tracking based on FCP. Study on the dynamic tracking method has the advantages of the high accuracy of defects and high real-time monitoring. So, the research of Java path test tracking method based on FCP is of great significance. In 2011, Zhang Rong from the Xi'an University of Posts & Telecommunications published an article[21] on the dynamic test of the white box. This paper described the use of dynamic testing techniques to obtain code coverage, program execution time information, but have not done path tracking study. In 2001, Nanjing university had developed a EASTT[31] system for C/C++ language, the dynamic test module of EASTT aimed at testing a specific case to analyze various levels of coverage under the procedures, including test efficiency, real-time coverage of function calling relationships *etc*. In 2009, Xi'an University of Technology published a paper[32] about embedded software path coverage and testing data collection, it designed a dynamic test data collection framework of embedded software, and completed the basic path test data acquisition and simple defect analysis. In 2008, Wang

ISSN: 1738-9968 IJHIT

Yue and others from the Beihang University, through the research of Java instrumentation and synchronization lock, put forward a dynamic tracking technology of concurrent programs based on Java instrumentation[33], can effectively track the execution of the thread. In 2006, Liu yongpo and others from the Beihang University studied Java program memory usage [34]. By monitoring the running status of the memory leak objects and the free object, the mode of constructing the behavior object is put forward, which can effectively reduce the use of the low efficiency memory and improve the efficiency of the system. This method was simple, high efficiency, but the function is limited, can only monitor the performance related issues. In 2006, Liu Zhenan and Zhang Qiang of University of Science and Technology of China put forward a kind of Java runtime exception analysis technology[35] based on class file. This technique inserts some exception handling code in the class file. When running the program to monitor the exception thrown and processing, gives a detailed analysis of the abnormal report, improve the efficiency of the treatment of loopholes. This method was fast, small overhead, and got more information, but the analysis ability was limited, can only find a few holes, and can't find the holes without exception. In 2013, Wang Chengjia from Huazhong University of Science and Technology in the direction of research on dynamic monitoring technology based on Java source code[20], similar to my study, but the research of Wang Chengjia based on sentence level. Most of the researchers in the study are based on the block of statements, the workload is very large, as the FCP based Java path test tracking method of high efficiency, low cost, high efficiency.

In 1995 May, Sun released Java, Java technology occupies a large proportion in the software development industry, it also brings great challenges to the efficiency and high accuracy of Java test. Java as an object-oriented language, with encapsulation, inheritance and polymorphism characteristics, the traditional testing technology can accurately and efficiently test. Software testing is an indispensable part of software project. And the software testing cost of the entire project 50%~75%, therefore, the research of Java function call tracking method for path test is very meaningful. In this paper, combined with Soot technology [3,5,6,7] and instrumentation technology [4, 21] to solve the practical problems of algorithm.

2. Definitions

The definition of FCP has already been defined in reference [9], the following definitions are used in this paper.

Definition 1: Dynamic path refers to input test data after instrumentation and run the program to get a series of feedback, then obtain dynamic FCP after analysis. Represented as: $L(S,t_1)=\{F_1 \cup F_2 \cup ... \cup F_n\}=\bigcup_{i=1}^{n} F_i$, S: The source program after instrumentation. t_i: the input test cases. F_n: function nodes of dynamic path.

Definition 2: Dynamic path set refers to run the test case set to obtain a collection of dynamic paths. Represented as: $L'=\{L(S,\bigcup_{i=1}^{n} t_i) \mid \forall t_i \in TestCase\}$,$TestCase$: the test case set.

Definition 3: The test case refers to a set of conditions or data, designed by tester based on program testing environment, to verify the effectiveness of program function. Represented as: $t_i =< E, L(S,D_i) >$. E: testing environment, $L(S,D_i)$: dynamic path D_i:test data.

Definition 4: Redundant test case refers to the test case input leads to dynamic path is repeated in dynamic path set. If $\forall t_i \in TestCase$,$\forall t_j \in TestCase$, $\exists(\mathrm{L}(\mathrm{S},t_i)=\mathrm{L}(\mathrm{S},t_j))$, then t_i or t_j is a redundant test case.

Definition 5: Repeated dynamic path refers to: $\forall \mathrm{t_i} \in \mathrm{TestCase}$, $\forall \mathrm{t_j} \in \mathrm{TestCase}$, $\exists(\mathrm{L}(\mathrm{S},\mathrm{t_i})=\mathrm{L}(\mathrm{S},\mathrm{t_j}))$. Contains dynamic path refers to input $\mathrm{t_i}$ to get a dynamic path as $\mathrm{L}(\mathrm{S},\mathrm{t_i})$, when run again to get a path as $\mathrm{L'}(\mathrm{S},\mathrm{t_i})$, if $\mathrm{L'}(\mathrm{S},\mathrm{t_i}) \subseteq \mathrm{L}(\mathrm{S},\mathrm{t_i})$, then $\mathrm{L'}(\mathrm{S},\mathrm{t_i})$ is

a contains dynamic path.

Definition 6: Single Linked List expressed as $\text{Link}(A_i) = \{A_i, < V_{i1}, \ldots . V_{in} >\}$, A_i: the current function, it's data structure is expressed as $< \text{key}, \text{link} >$. *Key* : the function signature, *link* : point to the address of called function, ; V_{in}: The function called by A_i. The data structure of called function is expressed as $\text{Call}(V_{in}) =< F_i, \text{Bool}, \text{link} >$, F_i : the called function. If Bool is 1, it has branch path. If Bool is 0, it has no branch path. All header functions are put in the array.

3. FCP Testing Process Tracking Algorithm

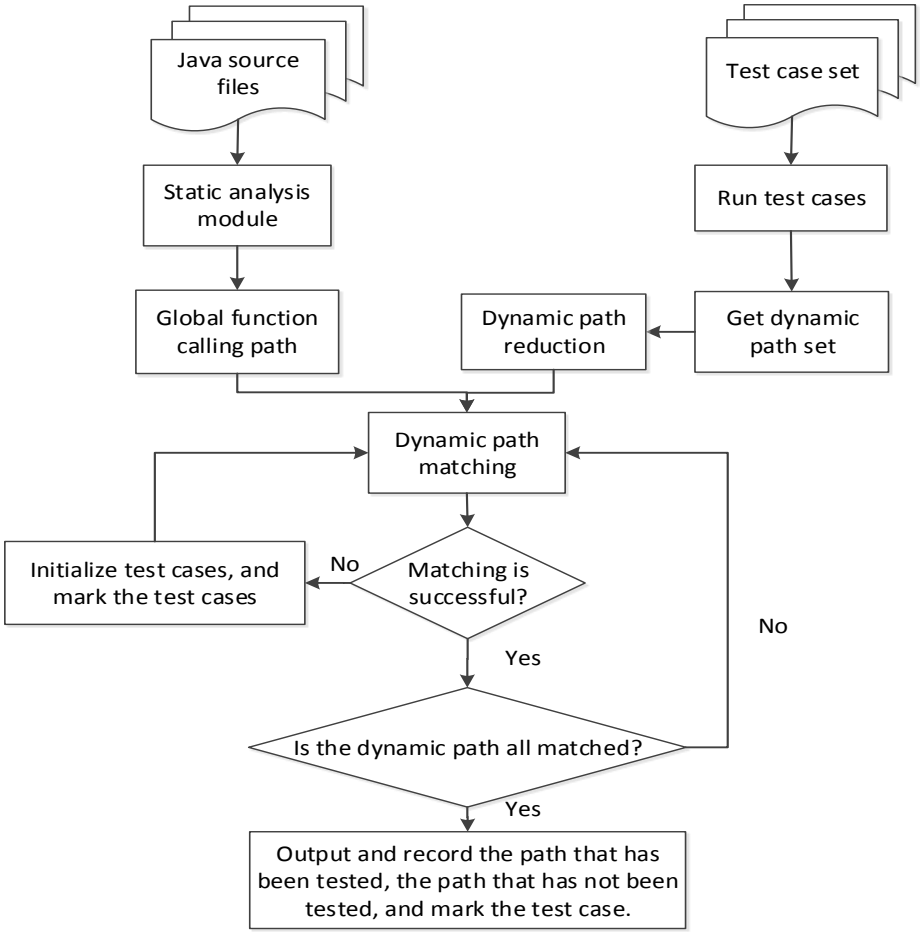

Figure 1 . FCP Test Process Tracking Model

3.1. FCP Generation Algorithm

In the previous work, the task group has done a lot of work[39,40] on the function calling path graph generation algorithm[4]. The work (including the local function calling path generation algorithm and the global function calling path generation algorithm) is to get a directed graph, which is composed of the calling relationship between functions, to facilitate the visualization of the call. It also provides a good data structure type - adjacency list for path matching.

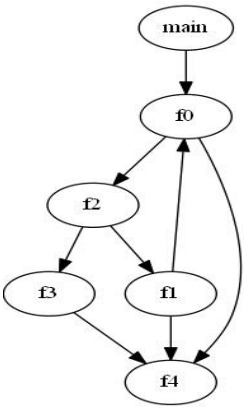

Figure 2 . Graph of FCP

First, get the class information- member function list through the Soot. Through the analysis of each member function N_i, get the execution statement of the N_i, and traverse each execution statement. If there is a function calling statement, get the information about the function to be called, and add to the local function calling graph. Then, find the list and this list is the main function for the header. Loop traversal each node N_i in the function calling graph of main function. Find whether the member function of current node N_i corresponds to a function calling graph. If there is such a function calling graph, the graph is incorporated into the function calling graph of the main function. Finally, through a combination of methods get the Java program global function calling graph. The single list header into the array, formed a function calling graph adjacency list. The data structure is shown in Figure 3:

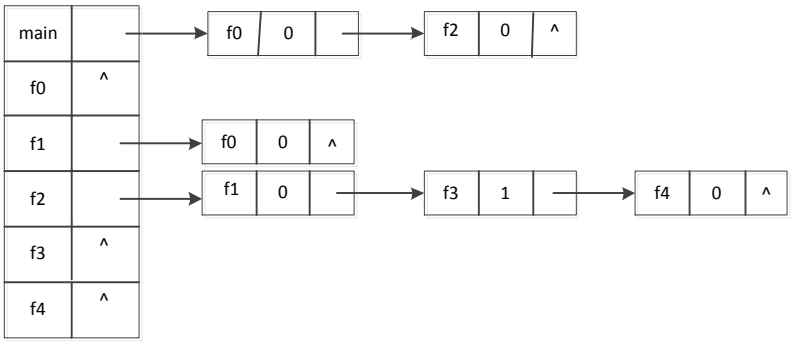

Figure 3 .The Data Structure of FCP

3.2. Repeat Dynamic Paths Reduction Algorithm

It has been mentioned many times to remove the repeated path, in fact, is to remove the redundant dynamic path. The dynamic path is actually a sequence of data stream consisting of instrumentation information. After the insertion, the data stream of all the dynamic paths will be obtained. Repeated dynamic paths reduction algorithm as described below:

a. all dynamic path sub sets are visited.

3.3. Contains Path Reduction Algorithm

In the test database, there are test data D_1, D_2, which makes $L(S, D_1) \neq L(S, D_2)$. But there is $L(S, D_1) \subset L(S, D_2)$, that is, $L(S, D_1)$ is a subset of $L(S, D_2)$,

$L(S, D_1)$ is a contains dynamic path. This requires the reduction of the contains path. For example, the implementation of test case T1 for word2013 print and the implementation of test case T2 for word2013 to select the "file" will get the path L1, L2. Then L2 is a contains path. This requires the removal of the contains path in the dynamic path set.

a. The path in the dynamic path sub set is divided into sub sets P0, P1, P2, …, Pn according to the length of the path. Set length (P0) < length (P1) ... <length (Pn).
b. From the definition of contains path we can know if path1⊆ path2, and length(path1) < length(path2), then path1 requires reduction. From the set Pk (k=0,1,…n) to take a path, to match the path in set Pm（m>k>0，Length(Pk)< Length(Pm)) by using LCS algorithm. If the length of the LCS is the same as that of the pathi, the pathi is a contains path. Remove contains path, and mark the corresponding test cases.
c. Until all of the path in the set Pk matching is completed, k=k+1, you need to continue to match the set P (k+1) to determine if there is a contains path. Repeat c.
d. Until m=n, at this point, the subset of the sub set will finish the reduction of contain path.

3.4. Obtain Dynamic Paths

The starting probe point is expressed as ProbeS, and the ending probe point is expressed as ProbeE. The expression as (ProbeS∪ProbeE)→F describes mapping relationship. If the control keyword is true, the probe point is expressed as ProbeT, else is expressed as ProbeF. Also there is a mapping relationship: (ProbeT∪ProbeF)→C. In order to FCP visualization, need to convert probe information into the corresponding function name.

3.4.1. Instrumentation of Information Storage

Running the instrumented program can collect information of probe points. The information stored in the form of a HashMap. To save the starting probe point s_n and the ending probe point e_n of every function as the value of HashMap, and the mapping function f_n as the key of HashMap. Moreover, to save the true and false probe points of the control keyword as the key of HashMap, and the control keyword as the key of HashMap. In order to facilitate the management of the information flow, the information returned by the list is stored in a list.
Combined with Soot and plug-in technology to get the function probe point mapping library as shown in table 2.1:

Table2.1 Function Name and Probe Point Mapping Library

Function name	Start probe point	End probe point	The starting line	The ending line
f1	00000001	1f000001	4	11
f2	00000002	1f000002	12	17
f3	00000003	1f000003	18	23
f4	00000004	1f000004	24	29
f5	00000005	1f000005	30	35
f6	00000006	1f000006	36	41
main	00000007	1f000007	42	71

3.4.1. The Algorithm of Obtaining Dynamic Paths

The algorithm of obtaining dynamic paths is described as follows:

Input: probe information: Probe ={ ProbeS∪ProbeE∪ProbeT∪ProbeF}。

output: dynamic path L。

Variable:

hashmap: Used to store instrumentation information. Convenient to 8 hexadecimal data stream sequence mapped to a function name to form the dynamic function call path.

arraylist: Used to store the mapping dynamic path

probe: The variables used to traverse the probe information

temp：To store function name 。

KeyWord: To store control keywords ,such as if，switch

Begin

do:

```
 set (arraylist);
 for each (probe∈ Probe)
    if(hashmap. containsKey(probe)) {
          temp=hashmap.get(probe);
          if(temp∈ KeyWord)    probe++;
          end if
          arraylist.add(temp)；  }
      end if
   end for
```

End

s_i is the entry point of function f_i. e_i is the exit point of function f_i. If the next probe information is not e_i, but it's the entry point of other function, indicating that f_i calls other function f_j. The function f_i itself is not over. The control keywords as instrumentation is to identify the calling relationships between functions. For example, in the dynamic path records the entry point of function f_1, the probe point of keyword if, entry point of function f_2. Note that the function F2 is included in the function F1, and the F2 execution is called by selecting the branch conditions for the true. When the loop structure and recursive structure calling function, we only process as a loop and a recursion.

According to the mapping relation of probe point data flow information with the function name, the probe point data flow information one one can be transformed into corresponding dynamic path consisting of the name of function.

3.5. Paths Tracking Algorithm

The algorithm monitors Java program dynamic execution by instrumentation technology. Then match the dynamic path and the FCP, in order to achieve the path tracking information. Most developers do not know how many unexpected paths will produce, which increases the risk of system vulnerabilities. From the perspective of developers, this algorithm greatly solve these problems. The following is the description of the path tracking algorithm:

Input: FCP graph G, Dynamic path set L

Output：Tracking results: result

Variable: Dpath: Storage dynamic path

visited: Tag function nodes with visited

p: The indicator to visit function nodes

adjlist：Adjacency list of G

Begin


```
do:
 for each (N_i ∈ G)
    initialize(visited[i]);
end for
for i←Dpath.Count-1 to 0
   for each (N_i ∈ G)
       if(!visited[i]&&G.v == Dpath[i])
visited[i]=TRUE;
            p=G->adjlist[i].firstedge;
while(p){
            if (visited[p->adjvex]&& G.v != Dpath[i])
                Recursive search p->adjvex;
                p=p->next;
              end if}
            end while
         end if
end for
end for
end
```

To compare function node F_i of dynamic path P_i and function node v_i of G, if F_i and v_i are the same, then tag is true, continue to compare, until the compare finished or the maximum depth of G is N; if F_i and v_i are not the same, trace back to the function node v_{i-1}, v_j is adjacent point of v_{i-1}. If v_j is visited, continue to compare v_{j+1}; If v_j is not visited then compare $F_i(F_i \in P_i)$ and v_j. If $\forall v_j$ is visited, trace back to v_{i-2}. Repeat the above process until back to the second level of the G. If the above procedure had been performed completely, but could not find the same path, matching failure. That is $P_i \neq (StaticPath_i \in \mathrm{FCP}(\bigcup_{i=1}^{n} StaticPath_i))$, dynamic path tracking failure, and to record and maintain test case library. If matching success, that is $P_i = \left(StaticPath_i \in \mathrm{FCP}(\bigcup_{i=1}^{n} StaticPath_i)\right)$, dynamic path tracking success.

3.6. Test Case Markers

In software testing, test case design is the core of the whole testing process, it is also the basis of software test execution process. The iterative development of the software makes the software testing personnel carry on round after round of regression testing, which makes the test case library huge, repetitive and non organization. Lead to the test case library is difficult to manage, increase the burden of testing personnel, reduce the efficiency of regression testing. So it is one of the hot topics in the research of software testing to explore the selection and maintenance of test cases. In this paper, a regression test case marking method is proposed, which is conducive to the timely and effective management of test case library.

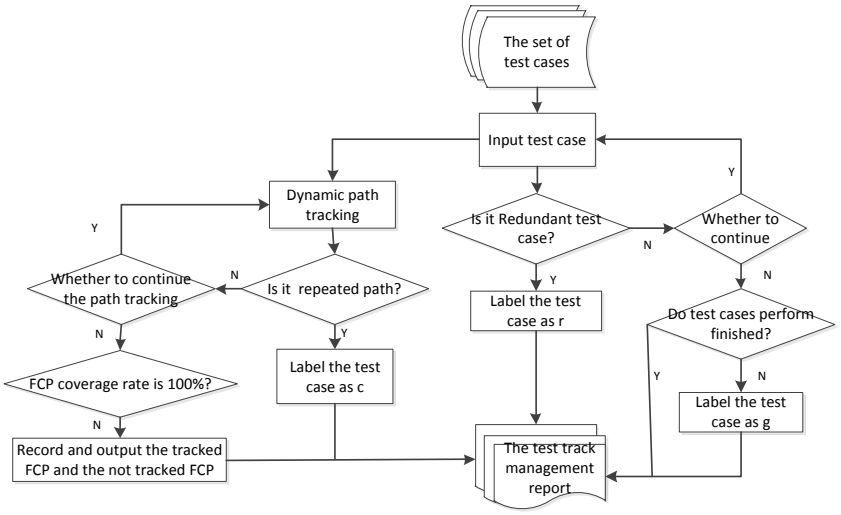

Figure 4 .Test Case Markers

Test case marks are marked mainly for the following:

（1） Mark the path has been tested;

（2） Mark the path has not been tested;

（3） Mark test cases interrupted during the test;

The mark of test case is mainly aimed at the mark of redundant test case and the mark of the test case not executed. Assuming that there are {N| (N>=2)} test cases, the dynamic paths that will be obtained after they are running are the same. There are (N-1) test cases are redundant. This requires the N-1 test cases to be marked, here we use r to mark. In the next regression test, we can avoid the test cases marked as r, and we can also choose to remove the redundant test cases labeled as r from the test case library. In another case, we mark a test case that has been passed for 1. If the test case is not tested at a certain time, we will mark the next test case that needs to be tested for g, to facilitate the next time testing starts with a test case marked as g.

4. Experiment and Evaluation

An example is given to illustrate the practicality and effectiveness of the Java function call path testing process tracking.


```
1. package com;
2. import java.util.Scanner;
3. public class hello {
4.         public static int f1(int i)
5.         {
6.                 System.out.println("f1\n");
7.                 int j;
8.                 if(i==1)j=1;
9.                 else j=i+f1(i-1);
10.                return j;
11.        }
12.        public static int f2 (int i)
13.        {
14.                System.out.println("f2\n");
15.                i = i+2;
16.                return i;
17.        }
18.        public static int f3(int i)
19.        {
20.                System.out.println("f3\n");
21.                i = i+3;
22.                return i;
23.        }
24.        public static int f4 (int i)
25.        {
26.                System.out.println("f4\n");
27.                i = i+4;
28.                return i;
29.        }
30.        public static int f5 (int i)
31.        {
32.                System.out.println("f5\n");
33.                i = i+5;
34.                return i;
35.        }
36.        public static int f6(int i)
37.        {
38.                System.out.println("f6\n");
39.                i = i+6;
40.                return i;
41.        }
42. public static void main(String[] args) {
43.        Scanner sc = new Scanner(System.in);
44.        int i=sc.nextInt();
45.        System.out.println("main\n");
46.        while(i<16)
47.        {
48.          if(i<4)
49.          i = f1(i);
50.          else (i<11)
51           {
52.            for(int k=3;k<6;k++)
53.            {
54.               if(k<5)
55.                   i=f2(i);
56.               else
57.                   i=f3(i);
58.            }
59.           }
60.          i=f4(i);
61.        }
62.        switch(i)
63.        {
64.          case 16:i=f4(i);break;
65.          case 18:i=f5(i);break;
66.          case 21:i=f6(i);break;
67.          default:
68.             System.out.println("Don't give up");
69.        }
70.      }
71. }
```

Figure 5 .Experiment Use Cases

4.1. The Instance Analysis

A total of 11 global FCP are generated by the function calling path generation algorithm, as shown below:

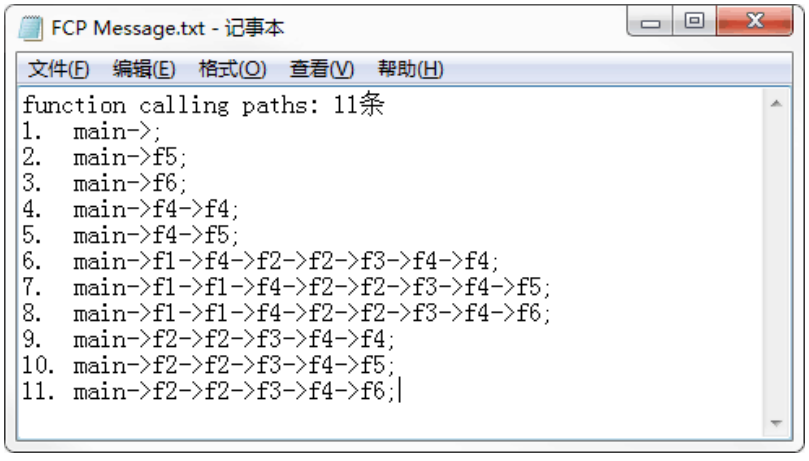

Figure 6 .Output of FCP

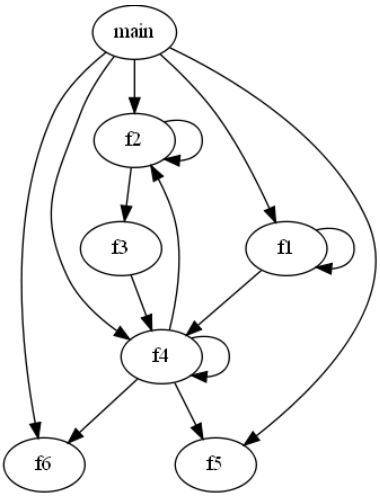

Figure 7 .Global Function Calling Graph

It can be seen from the results that the algorithm accurately analyzes the call relationship between functions, and gets the global function calling graph. The result is consistent with artificial static analysis.

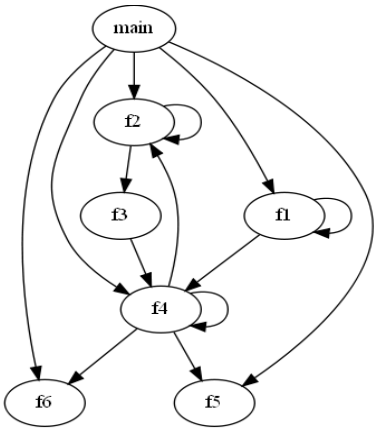

Figure 8 . Graph of FCP

Enter test data for test case set {i=1; i=4; i=5; i=8; i=10; i=11; i=13; i=15; i=18; i=22;}, get the data instrument point stream sequence of the dynamic path of the program, and then through the dynamic path reduction algorithm to remove redundant dynamic path. Finally, the dynamic path is extracted to obtain the dynamic path of the program. Input test data as i=18, we can get the path: main->f5. Tested path is displayed as the dotted line, it's convenient to the visualization of the testing process. Other test data is also based on this method to get the corresponding visual FCP. The visual test tracking of the test data i=18 is shown in Figure 9.

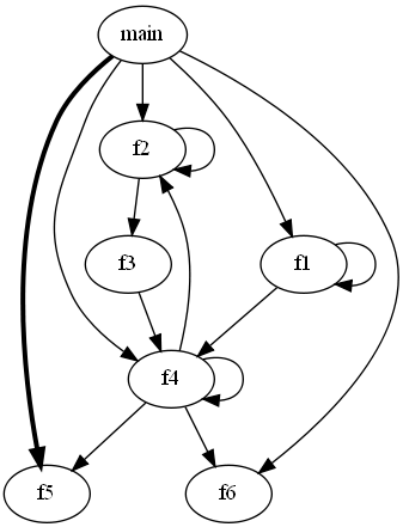

Figure 9 The Test Tracking Result Where I=18

The next step, statistics derive from tracing FCP. Match the dynamic path with the global FCP that has been acquired. Get the number of paths that have been tracked and recorded the effective test cases, and then get the path and the number that have not been tracked. Finally, to record the log of test cases for regression testing. The results are shown in Figure 10:

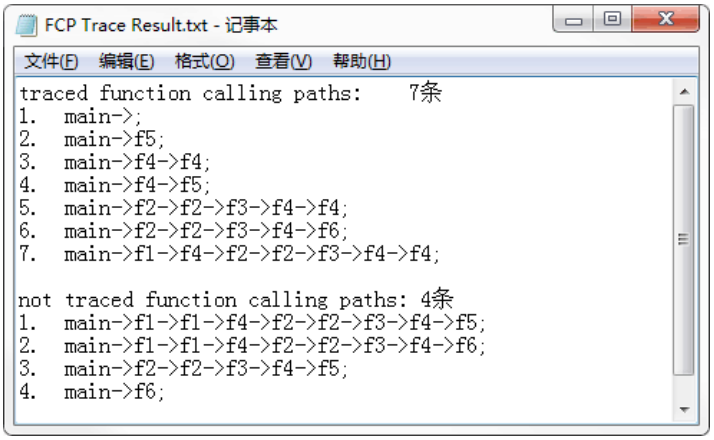

Figure 10 .The Statistics of Tracking Result

The test case set is marked as follows:

Table 2.2. Test Case Markers

The code of test case	Test case	Test path	Test case marking	Path marking	Record
1	i=1	main->f1->f4->f2->f2->f3->f4->f4	1	1	A total of 11 reduction FCP, which has 7 FCP has been tested and 4 FCP not tested
2	i=4	main->f2->f2->f3->f4->f4	1	1	
3	i=5	main->f2->f2->f3->f4->f4	c	r	
4	i=8	main->f2->f2->f3->f4	1	1	
5	i=10	main->f2->f2->f3->f4->f6	1	1	
6	i=11	main->f4->f4	1	1	
7	i=13	main->f4	1	1	
8	i=15	main->f4	c	r	
9	i=18	main->f5	1	1	
10	i=22	main->	1	1	

In the experiment, there is a need to increase the design test cases for 4 paths which are not tested. As shown below:

Table 2.3. New Test Case Markers

The code of test case	Test case	Test path	Test case marking	Path marking
1	i=1	main->f1->f1->f4->f2->f2->f3->f4->f5	a	0
2	i=4	main->f1->f1->f4->f2->f2->f3->f4->f6	a	0
3	i=5	main->f2->f2->f3->f4->f5	a	0
4	i=8	main->f6	a	0

4.2. The Experimental Results

In order to prove the efficiency and accuracy of this algorithm, the input source program files, the results are shown in Table 2.4. The time in the experiment is the time when the corresponding number of lines of code to complete the test.

Table 2.4. The Experimental Results

Lines of code	Time-taken (ms)	The number of Global FCP	The number of the traced paths
61	10	13	8
500	80	22	13
1000	300	34	15
5000	500	50	19
10000	2000	89	27

From the above table can be seen, the algorithm can quickly and accurately track. It can be applied in most of the test work. But in the face of large and complex system code, tracking efficiency decreased. This shows that the algorithm has some areas to be improved, which will be the next phase of the research objectives.

5. Conclusion

Path testing is a hot issue. Especially in large-scale software, to complete path test is nearly impossible, and the time is considerable to test by manual. So it greatly reduces the difficulty of the path tests based on the FCP. Experiment show that the research of this thesis can accurately track path, match the dynamic path and FCP, then display the tracked paths and the no tracked path. And, it can record test cases for regression testing. So, this makes the research of the object-oriented language function call path test tracking no longer a theoretical idea, but a practical approach to the field of software testing and development.

References

[1] B. Beizer, Software Testing Technique》, Van Nostrand.
[2] M. Weiser, "Programmers use slices when debugging", Communications of the ACM, vol. 07, **(1982)**.
[3] Z.-H. Zhang, M.-Min, "Research of Path Coverage Generation Techniques Based Function Call Graph", Acta Electronicasinica, vol. 38, no. 8, **(2010)**, pp. 1808-1811.
[4] Y. Mu, Z. Jiang and Z. Zhang, "Path extraction based on C program instrumentation", Computer Engineering and Applications. TP311，1002-8331（**2011**）01-0067-03.
[5] A.B. Einarsso and J. D. Nelision, A Survivor's Guide to Java Program Analysis with Soot, **(2008)**.
[6] Lhoták Ond ej.Spark, "A flexible points-to analysis framework for Java", Montreal: School of Computer Science, McGill University, **(2003)**.
[7] R. Vallee-Rai, P. Co and E. Gagnon, "Soot: a Java Bytecode Optimization Framework [EB/OL]", (1999-06-24).
[8] X.-H. Xing and H. Liu， "Research on LCS Base Chinese Abbreviation Field Match"，Shangdong Sciences，vol. 21，**(2008.8)**, pp. 52-56.
[9] M. Yan, Y. Mu, Y. He and A. Liu, "The Analysis of Function Calling Path in Java Based on Soot", Applied Mechanics and Materials, vol. 568-570, **(2014)**, pp. 1479-1487.
[10] Y. L. Li, H. Chen and L. Liu, "Java Program Control Flow Analysis and Graphic Output of Soot", Computer Systems & Applications, vol. 10, **(2009)**.
[11] X. Zhu，Y. M and Z.H. Zhang, "Analysis of function call path based on control flow graph by Soot", Data Communications，**(2012.4)**.
[12] Y.M. Mu, Y. Ding and Z.H. Zhang, "Research on Determination of Class Template Call Path Uniqueness for Regression Testing", Tsinghua Science and Technology，**(2012)**，vol. 52, no. 1, pp. 33-39.
[13] Y.-M. Mu, J. Yu and Z.-H. Zhang, "Research of the Hot Paths in Software Automation Testing", Computer Engineering&Science，vol. 33, no. 6, **(2011)**.
[14] Y. Wang，T.-H. Jiang，J. Don and X. Zhou, "Implementation of Executable Code Test Tools Based on Path Coverage Instumentation", Computer Engineering，vol.38, no.5, **(2012.03)**.
[15] M. Yan, Y. Mu, Y. He and A. Liu, "The Analysis of Function Calling Path in Java Based on Soot", Applied Mechanics and Materials, vol. 568-570, **(2014)**, pp.1479-1487.
[16] L. Yu, "Software testing method and practice", Beijing: Tsinghua University press, vol. 11, **(2008)**, pp. 8-70.
[17] Y. Zhang and Y.Y. Liu, "Java Bytecode Optimization Framework", Computer Engineering, vol. 34, no.2, **(2008)**.
[18] http://www.massapi.com/source/sootsrc2.4.0/src/soot/tools/CFGViewer.java.html.
[19] Y. Mu, Y. Zheng, Z. Zhang and M. Liu, "The Algorithm of Infeasible Paths Extraction Oriented the Function Calling Relationship", Chinese Journal of Electronics, vol.21, no.2, **(2012)**, pp.236-240.
[20] C. Wang, "Research on Dynamic Monitor System based on the Source Code of Java", Huazhong University of Science and Technology, Wuhan 430074, P. R. China .January, **(2013)**.
[21] R. Zhang and S. Wang, "Research and Implementation of Dynamic Tests Based on Instrumentation Technology", Modern Electronics Technique. vol. 34, no.4, **(2011)**.
[22] D. Shi, "Computer Measurement ＆ Control", Computer Measurement ＆ Control[m], vol. 18, no. 10, **(2010)**.
[23] B. P. A. J. Nievergelt, "Monitoring Program Execution: A Survey", Computer, vol.1981, vol. 14, pp. 76-93.
[24] N. Halbwachs, L. Zuck and F. Chen "Java-MOP: A Monitoring Oriented Programming Environment for Java", Springer Berlin Heidelberg, vol. 3440, **(2005)**, pp. 546-550.
[25] D. Cofer, A. Fantechi and C. Colombo, "Dynamic Event-Based Runtime Monitoring of Real-Time and Contextual Properties", Lecture Notes in Computer Science .5596: Springer Berlin Heidelberg, **(2009)**, pp. 135～149.

[26] A. Seesing and A. Orso, "InsECTJ: a generic instrumentation framework for collecting dynamic information within Eclipse", in: San Diego, California: ACM, **(2005)**, pp.45-49.
[27] B. Meyer, J. Woodcock and K. Havelund, "Verify Your Runs. Lecture Notes in Computer Science", 4171: Springer Berlin Heidelberg, **(2008)**, pp. 374-383.
[28] R. Alur, D. Peled and C. Artho, "JNuke: Efficient Dynamic Analysis for Java", Lecture Notes in Computer Science, 3114: Springer Berlin Heidelberg, **(2004)**, pp. 462-465.
[29] O. Sokolsky, K. Havelund aand I. Lee, "Introduction to the special section on runtime verification", International Journal on Software Tools for Technology Transfer, vol. 14, no. 3, **(2012)**, pp. 243-247.
[30] M. Kim, M. Viswanathan and S. Kannan, "Java-MaC: a run-time assurance approach for Java programs", Formal Methods in System Design, vol. 24, no. 2, **(2004)**, pp. 129-155.
[31] C. Rosenkranz M. C. Charaf and R. Holten, "Language quality in requirements development: tracing communication in the process of information systems development", Journal Inf technol, **(2013)**.
[32] Q.-X. Yu, Y.-K. Zhang, Y.J. Hu and W. Zhu, "Data Collection in Embedded Software Path Coverage Test", Computer Engineering, vol. 21, **(2009)**, pp. 54-56.
[33] X.-P. Zhang, Baodan and J.-Y. Wang, "Research on Testing Strategies and Methods for Java Multithread", Application Research of Computers, **(2006)**, vol. 1, pp. 12-14.
[34] Y.-P. Liu, X.-X. Jia, J. Wu, M.-Z. Jin and H.-L. Sun, "Analysis of Memory Usage with Low Efficiency in Java Program", Computer Engineering, **(2008)**, vol. 23, pp. 84-85.
[35] Z.-A. Liu and Q. Zhang, "An Analytical Technique of Java Program Runtime Exception Based on Class Document. Measurement & Control Technology", , vol. 11, **(2006)**, pp. 61-63.

