2011 International Conference on Computer Science and Information Technology (ICCSIT 2011)
IPCSIT vol. 51 (2012) © (2012) IACSIT Press, Singapore
DOI: 10.7763/IPCSIT.2012.V51.56

An Approach to Generate Basis Path for Programs with Exception-Handling Constructs

Qingtan Wang[+], Shujuan Jiang and Yanmei Zhang

School of Computer Science and Technology, China University of Mining and Technology
Xuzhou, 221116, Jiangsu Province, China

Abstract. Exception handling can improve software robustness. However, it increases the cost of software testing. Basis path testing is a simple and efficient white-box testing method. Traditional research of basis path testing ignored the impact of exception handling on control flow. This paper gives an algorithm to generate basis path set (BPS) based on Exception Control Flow Graph (ECFG). Meanwhile, we apply the method to a case study. The results show that our approach is efficient.

Keywords: exception-handling constructs; basis path testing; ECFG

1. Introduction

Software testing is divided into black-box testing and white-box testing. It is very important in the software development cycle. In fact, no matter which method is used, it is impossible to realize complete testing. McCabe[2] proposed a method called basis path testing, which is based on vector space. Facts have proved that this method can detect defects with less cost [11]. Current study of basis path testing is mainly based on C language, which doesn't contain exception. There is few research about C++, Java and Ada. The control flow graph of program with exception handling, constructed by the traditional method, isn't entirely correct.

Exception handling is an important way to improve software robustness, and in real program, it accounts for about 10% of the total function [6,7]. When there are any predictable errors, the program itself is expected to take some rational behaviors. This is the reason why the exception handling appeared. The task of exception handling is to transfer the control from the place where exception occurs to exception handler which can handle the exception. When an exception occurs and some operations are not executed, the program should be able to catch the exception and take appropriate measures. The program can choose to return to a safe state and allows the user to perform some other commands, or allow the user to save all results, and terminate the program in an appropriate manner. Because of the flexibility of the exception and programmers' neglect of exception handling, the exception handling often leads to defectsm [9].

In current research, the testing of program with exception handling constructs is always separated into normal testing and exception testing [5,8,9]. So we need to test one program twice separately which increases the cost of testing. Aiming at these problems, this paper proposes a method to detect normal defect and exception defect, performing only once basis path testing.

The rest of the paper is organized as follows. Section 2 reviews some terminologies that will be used in this paper. Section 3 presents an algorithm of constructing the BPS. Section 4 shows a case study. Section 5 discusses the related works and Section 6 concludes the paper.

2. Background

[+] Corresponding author. *E-mail address: wang_qingtan@163.com.*

In this section, we describe the concepts used in this paper and gives a brief description of the exception handling mechanism in Java language.

When testing a program with exception handling constructs, ECFG is used to represent the structure of the program. ECFG is a directed graph, represented as: ECFG=<N,E,Entry,Exit> , where N is a set of nodes, E is a set of edges, Entry is the unique entry node and Exit is the unique exit node. Each node represents a program statement or an exception exit. Each edge (m, n) represents the flow of control from statement m to statement n [10].

A BPS is a set of mutually independent paths through which all the paths can be constructed by linear computation [11]. The number of the BPS equals to the number of Cyclomatic complexity (CC), which is defined to be $e - n + 2$, where e and n are the number of edges and nodes in the program flow graph, respectively.

In Java, exception is encapsulated into class. All exception classes are subclasses of java.lang.Throwable directly or indirectly. Exception objects can be thrown by Java virtual machine in runtime or by throw statement in a program. When an exception object is thrown, we can use the try-catch-finally structure to catch it or throw it to the outer method.

3. The application of ECFG in basis path testing

3.1. Construcion of ECFG

The main task of exception handling is to transfer the control from the place where exception takes place to the exception handler. If the effect of exception handling on the control flow is not taken into account when constructing control flow, the control flow graph is not able to represent the real control flow. This paper makes use of the algorithm proposed in reference [7] to construct the ECFG which includes exception handling constructs. There are three steps: First, construct an uncompleted control flow graph in which there is not outgoing edge matching to the statement of throw; second, determine the exception type of the statements that may thrown; finally, add the outgoing edges of the statement of throw and the necessary exception nodes. The details are in reference [7].

3.2. Construcion of ECFG

The main work is to obtain the BPS after constructing the ECFG. In this subsection, we propose an algorithm to construct BPS automatically, which is shown as Figure 1.

```
Algorithm: BPS-construction
Input: G(P)//CFG or ECFG
Output：BPS
begin
1 if G has loop then
2   EdgeList={ e|∀e∈G ∧G-e doesn't include loop}
3   G'=G-EdgeList
4  else G'=G
5  endif
6 BranchNodeList={n|∀n∈G'∧OutOf(n)>1}
7 NeedAnalysisPath = {path| path is got from G' by DFS}
8 while(NeedAnalysisPath!=φ)
9   for each path in NeedAnalysisPath do
10     if (path∩BranchNodeList)!=φ then
11       for each node in path∩BranchNodeList do
12         WorkList= WorkList∪{<node,path>}
13         BranchNodeList= BranchNodeList-{node}
14       endfor
15      NeedAnalysisPath= NeedAnalysisPath-path
16      BPS= BPS∪{path}
17     endif
18  endfor
19  if WorkList!=φ then
20   for each <node, path> in WorkList do
21     for each other node's successor do
22       get new path p
23       NeedAnalysisPath= NeedAnalysisPath∪{p}
24     endfor
25     WorkList= WorkList-<node, path>
26   endfor
27  endif
28  for each edge in EdgeList do
29    get sp1 from Entry to edge's target node
30    get sp2 from edge's source node to Exit
31    NeedAnalysisPath= NeedAnalysisPath∪{sp1+sp2}
32  endfor
33 endwhile
34 return BPS(P)
end
```

Fig. 1 The construction algorithm of BPS

The main idea of the algorithm is: First, simplifies the graph to a simple graph without any loops (lines 1-5); then generates the BPS based on the simple graph (lines 6-27); finally, handles the loop structures (lines 28-32). In order to reduce the times of traversal, records the nodes whose outdegree is bigger than 1 into BranchNodeList, then does further analysis for those paths which include the nodes in BranchNodeList. What's more, in order to improve the efficiency, the nodes which need to be flipped are recorded.

4. Case study

In this section, to confirm the validity of the proposed approach, we apply the method to an example program. It is shown in Figure 2.

```
1      import java.util.Scanner;
2
3      public class SceondTest {
4          public static void main(String[] args) {
5              int opt, a, b, temp,r,max,min;
6              boolean next = true;
7              Scanner sc = new Scanner(System.in);
8               System.out.println("please input two  integers:");
9              a = sc.nextInt();
10             b = sc.nextInt();
11             if (a > b) {
12                 max = a;
13                 min = b;
14             } else {
15                  max = b;
16                  min = a;
17             }
18              System.out.println("options: \n1 get max and
                        min;"\n2 get greatest common
                        divisor;\nother exit");
19             opt = sc.nextInt();
20             try {
21                 switch (opt) {
22                 case 1:
23                     System.out.println("max is " + max);
24                     System.out.println("min is " + min);
25                     break;
26                 case 2:
27                     int t1,t2;
28                     t1 = max;
29                     t2 = min;
30                     if (t1 == t2)
31                         throw new MyException();
32                     int t3 = t1 % t2;
33                     while (t3 != 0) {
34                             t1 = t2;
35                             t2 = t3;
36                             t3 = t1 % t2;
37                     }
38                      System.out.println("greatest common
                              divisor is " + t2);
39                     break;
40                 default:
41                     break;
42                 }
43             } catch (ArithmeticException e) {
44                     System.out.println("Error! Can't be zero!");
45             } catch (MyException e) {
46                     System.out.println("Two numbers
                            equal!\nThe greaest common divisor is
                            " + max);
47             } finally {
48                     System.out.println("Analysis end!");
49             }
50         System.out.println("Program end!");
51     }
```

Fig. 2 example program

During the analysis of the example program, first, we construct two different control graphs using these two different methods. Then compute the basis path set using the algorithm proposed in this paper.

When not taking the effect of exception handling into account, the control flow graph (CFG) is shown in Figure 3(a). However, when considering exception-handling constructs, the corresponding exception control flow graph (ECFG) is shown in Figure 3(b). The numbers inside the nodes represent the line number of statements. After comparison, it is found that ECFG can cover all the statements and all the possible control transferring, which reflects the real program.

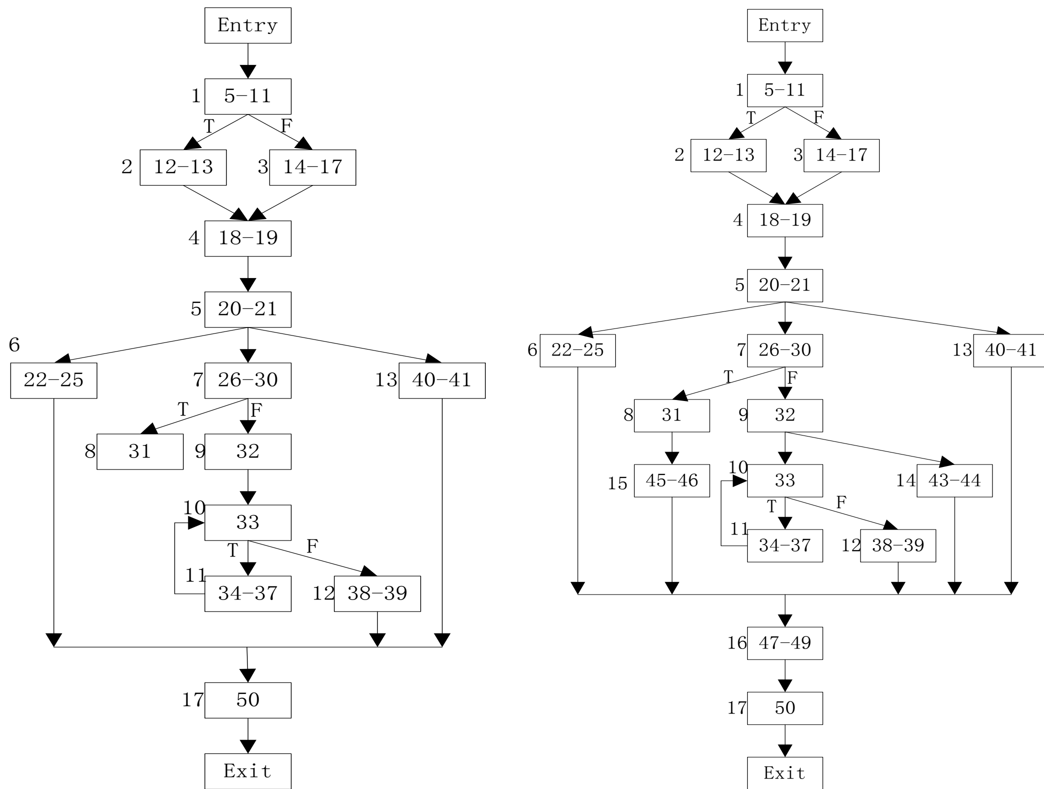

Fig. 3. (a) CFG of example program; (b) ECFG of example program

Taking the CFG and ECFG as the inputs of the proposed algorithm respectively, the results of the basis path set are shown in Table 1.

Table 1. BPS of CFG and ECFG

		CFG	ECFG
CC		19-16+2=5	24-19+2=7
BPS	Ⅰ	Entry-1-2-4-5-6-17-Exit	Entry-1-2-4-5-6-16-17-Exit
	Ⅱ	Entry-1-2-4-5-7-9-10-12-17–Exit	Entry-1-2-4-5-7-9-10-12-16-17-Exit
	Ⅲ	Entry-1-2-4-5-7-9-10-11-10-12-17–Exit	Entry-1-2-4-5-7-9-10-11-10-12-16-17-Exit
	Ⅳ	Entry-1-2-4-5-13-17-Exit	Entry-1-2-4-5-13-16-17-Exit
	Ⅴ	Entry-1-3-4-5-6-17-Exit	Entry-1-3-4-5-6-16-17-Exit
	Ⅵ	\	Entry-1-2-4-5-7-9-14-16-17-Exit
	Ⅶ	\	Entry-1-2-4-5-7-8-15-16-17-Exit

As shown in Table 1, compared with the BPS of CFG, we can find that there increases two paths in ECFG: VI and VII. This is because we take the effect of exception-handling in ECFG on the control flow. What’s more, the basis paths in ECFG include all the codes in the program, which implements complete converges of the codes.

5. Related work

The existing researches of testing program with exception-handling constructs mainly concentrated on defects brought by exception handling. Sinha et al[8] proposed a series of testing standards. These standards were proved that they were able to meet the testing demands. The method proposed in this paper meets a part of those standards, such as: all-e-defs, all-e-acts and all-e-deacts. Weimer[9] made use of data flow analysis to find the defects brought by exception handling. However, it is aimed at a given failure model for specific defects, rendering it could not be used widely. The method proposed in this paper, which is based on ECFG of the program, is able to be used widely and not limited to the models.

There are mainly two methods to generate basis path set: one is Baseline method proposed by McCabe[2], without algorithm of generating BPS; the other one is Depth Finding Strategy proposed by Poole[11], which is implemented by recursion. In the case of control flow graph being complicated, the memory overflow is easy to take place. As a result, it is not quite suitable for automatic testing. The method proposed in this paper combines the advantages of the above two methods. First, determine the baseline path by Depth Finding Strategy; second, obtain the BPS by flipping branch nodes. As a result, it is easy to handle complicated control flow graph, and can generate the basis path set automatically.

6. Conclusion and Future work

Exception handling increases the cost of software testing while improving the robustness of software. This paper mainly proposes a method to construct BPS, based on the ECFG. And the experiment shows the efficiency. However, the obtained basis paths are not all feasible. Our future work is to find out the infeasible basis paths, reducing the time of software testing.

7. Acknowledgements

This work was supported in part by awards from National Natural Science Foundation under 60970032, the Key Project of Chinese Ministry of Education under 108063, Natural Science Foundation of Jiangsu Province under BK2008124, Qing Lan Project, and Graduate Training Innovative Projects Foundation of Jiangsu, China under CX10B_157Z.

8. References

[1] P. Jorgensen, “Software Testing: A Craftsman's Approach,” 2ed. Boca Raton, FL: CRC Press, 2002.

[2] T. J. McCabe, “Structural Testing: A Software Testing Methodology Using the Cyclomatic Complexity Metric [M],” 1982. NIST Special Publication 500-99, Washington D. C.

[3] G. Wang, X. Jing, and Y. Wang, “The Application of Improved McCabe Method in Basis Path Test,” Journal of Harbin university of science and technology, vol.15, no.1, 2010, pp.48-51.

[4] Q. F. Du, and N. Li, “White Box Test Basic Path Algorithm,” Computer Engineering, vol.35, no.15, 2009, pp.100-102.

[5] C. Y. Mao, and Y. S. Lu, “Study on the Analysis and Testing of Exception Handlings in C++Programs,” Mini-micro Systems, vol.27, no.3, 2006, pp.481-485.

[6] M. P. Robillard, and G. C.Murphy, “Static analysis to support the evolution of exception structure in Object-Oriented systems,” ACM Transactions on Software Engineering and Methodology, vol.12, no.2, 2003, pp.191-221.

[7] S. Sinha and M. J. Harrold, “Control flow analysis of programs with exception-handling constructs”, Tech. Rep. OSU-CISRC-7/98-TR25, The Ohio State University, 1998.

[8] S. Sinha, and M. J. Harrold, “Criteria for testing exception-handling constructs in Java programs,” Proceedings of the International Conference on Software Maintenance, Los Alamitos: IEEE Computer Society Press, pp.265-274, 1999, Washington, USA.

[9] W. Westley, “Exception-Handling Bugs in Java and a Language Extension to Avoid Them,” Springer-Verlag Berlin Heidelberg. LNCS 4119, 2006, pp. 22-41.

[10] S. J. Jiang, B. W. Xu, and L. Shi, “An approach of data-flow analysis based on exception propagation analysis,” Journal of Software, vol.18, no.1, 2007, pp.74-84.

[11] J. Poole, “A method to determine a basis set of paths to perform program testing,” (NISTIR 5737), Tech. rep., Department of Commerce,NIST, Nov. 1995.