Infeasible Basis Paths Detection of Program with Exception-Handling Constructs

Yanmei Zhang, Shujuan Jiang, Qingtan Wang, Xuefeng Zhao
School of Computer Science and Technology, China University of Mining and Technology
Xuzhou, 221116, Jiangsu Province, China, ymzhang@cumt.edu.cn

Abstract

Infeasible paths increase the complexity and redundancy to programs. It is a key problem of structural testing to detect the infeasible paths. The paper proposes an infeasible basis paths detecting method for the program with exception constructs based on the correlations of different conditional statements. The technique combines exception propagation with the correlations of different conditional statements to analyze the impacts of exception propagation on paths feasibility. The correlations are determined according to dataflow analyses. Finally, we apply the proposed method in some programs. The results show that the method can accurately detect infeasible basis paths, which can save test resources and improve test efficiency.

Keywords: *Infeasible Paths, Exception Propagation, Paths Feasibility, Dataflow Analyses*

1. Introduction

Structural testing is an important part of many software engineering activities. However, the difficulty is the existence of infeasible paths in a program. The feasibility of paths tends to influence the efficiency and sufficiency of the structural testing directly. If there is no input for making a target path to be executed, the path is considered infeasible [1]. The cause and effect of infeasible paths of a program were discussed early by Hedley [1]. In addition, Bodik et al. [2] pointed out that the main cause of infeasible paths of a program was the correlation of some conditional statements. They also proved that 9-40% of conditional statements in a complicated program had correlations. Therefore, the correlations of conditional statements play an important part in detecting infeasible paths. The infeasible paths increase the cost of software testing and impacts accuracy of testing seriously. If a majority of infeasible paths can be detected during static analysis, it will greatly improve the performance of many software engineering activities, particularly structural testing and coverage analyses [3].

In path testing, the execution path of a program is often very complex. Test resources and test time is limited, so it is difficult to test all the paths. Meanwhile, program statements and branches can be covered through testing the basis path set, which can effectively enhance the test efficiency. Therefore we only need to detect the feasibility of paths in a basis path set. Meanwhile, with the introduction of exception-handling constructs in program, original execution paths are changed. Therefore, the influence of exception propagation on data flow also plays an important part in detecting infeasible paths.

In this paper, we propose an approach to detect infeasible basis paths for the program with exception-handling constructs by determining conditional statements correlations. This method combines exception propagation with the branch correlations of different conditional statements to analyze the impacts of exception propagation on paths, and determines the branch correlations of different conditional statements according to dataflow analyses.

The organization of the paper is as follows. Section 2 introduces a basis path set generation method. Section 3 proposes an approach to detecting infeasible basis paths for programs with exception constructs. Section 4 shows a case study. Section 5 gives some experiments. Section 6 reviews the related works on infeasible basis paths detection. Finally, section 7 draws conclusions and points out future research work.

2. Basis path set generation method

In this section, we introduce some related conceptions on infeasible path and a typical generation method of basis path set [4].

2.1. Related conceptions

Program path is a sequence of statements that are executed sequentially. It can be defined as follows:

Definition 1 Independent path: It means the path that at least contains one edge that never is included in other paths.

Definition 2 Basis paths set [5]: if any path P of a program's control flow graph can be linearly represented by *B,* then a linearly independent path set *B* of the program is said to be a basis path set of the program.

The main cause of feasibility problem is the existence of branched constructs. The branch judgment results directly impact the execution paths. Conditions predicates are always take the value of *true* or *false* in an execution path. Different predicates' outcome (i.e. *true* or *false*) causes the branch conflict, which leads some paths can not be covered. Based on the above analysis we can give the following definition.

Definition 3 Infeasible path [11]: Suppose n_i and n_j are two condition statements of P. If (n_i, n_j) have True→True (or True→False) correlation, then any path including the true branch of n_i and the false (or true) branch of nj will be infeasible; similarly, if (n_i, n_j) have False→True (or False→False) correlation, then any path including the false branch of n_i and the false (or true) branch of n_j will be infeasible. Where, the classification of correlation is specified on reference [11].

Generally speaking, if there is an input data that make a path executable, the path is feasible; the path that can not be executed by the program in any situation is infeasible.

2.2. Basis path set

Basis path set was first presented by McCabe [4]. He has proved that the number of basis path set is equal to cyclomatic complexity V(G). Therefore, we should determine cyclomatic complexity V(G) first before determining the basis paths.

Definition 4 Control flow graph (CFG): It is represented with a triad CFG=(N,E,N_0), CFG is a digraph, N represents a vertex-set of all vertexes in the graph, representing computational statements or expressions, E represents a edge-set of all directed edges in the graph, representing transfer of control between nodes, N_0 represents first vertex.

CFG is a graphic interpretation of describing program control flow; it describes the logic constructs of software modules. A module corresponds to a single function or subroutine in typical languages, has a single entry and exit point [4].

Calculation methods of cyclomatic complexity V(G) in basis path testing is [4]: V(G) =F+1. Where, 'F' denotes the number of lozenge judgment nodes in a CFG.

Take the situation of single entry & exit only as the example, the value of V(G) equals to the number of judgment statements plus 1.

It is necessary to point out that the path which is simply combined by existing paths is not an independent path.

All of the basis paths in a program can be obtained by the typical baseline method in reference [4].

3. Method of detecting infeasible path for programs with exception constructs

In this section, we present a method to detect the feasibility of paths for programs with exception constructs. We can convert the problem of infeasible path detection for programs with exception-handling constructs into the problem that judge the conflict of different branches using dataflow information. First, calculate variable definition-reach set OUT_DEF and use-reach set OUT_USE for each basis block using dataflow analyses technique; then in variables definition-reach set and use-reach set of the selected basis block, judge whether the conditional predicate of a basis block would get a fixed value through variable' definition point and use point; next, analyze the correlations of different branches, and according to the basis blocks where variable' definition point, use point and corresponding branches are, determine the correlations of two basis blocks; finally, the feasibility of the generated path set are detected using the correlations of different basis blocks.

3.1. Basis block

Basis block is a division of a program. It is a set of program statements that are executed sequentially. Each basis block contains a unique start statement and a unique end statement. From the concept of basis block, it can be infer that each basis block contains only one conditional branch statement. When a program is executed, its statements can only entry into the CFG from the entrance statement of a basis block and exit from the exit statement. For a given program, it can be divided into a series of basis blocks.

According to the characteristics of CFG and basis block, we can take basis blocks as the nodes of CFG.

The original execution paths may be changed due to exception propagation. Therefore, when a program contains exception-handling constructs, we need to consider all possible paths caused by exception propagation. Our method is to introduce the exception-handling constructs into the basis block level. The first statement of exception-handling module is used as the entrance statement of its corresponding basis block. Basis block is consists of the statements from entrance statement to the *throw* statement (including the *throw* statement). For the basis block where throws exception statements are, it has a directed edge which points to another basis block where corresponding exception-handling module is. Where, throwing ''exception'' represents that exception-handling branches need to be executed. The number of basis block is increased when taking the exception-handling modules as basis blocks.

We can judge the correlations of different conditional statements through dataflow analysis. Dataflow analysis is the process of dataflow information collection for data definition, data use and data dependency. Dataflow analysis technology is a relatively mature technology. Exception-handling is one of the mechanisms of most object-oriented programming language. As representative of object-oriented programming language, Java and C++ language both provide perfect exception-handling mechanism.

3.2. Calculation of definition-reach set and use-reach set for basis blocks

In this section, we adopt the dataflow analyses method to analyze the correlations of different branches according to definition-reach information and use-reach information of each basis block in the CFG. The definition information and use information of each basis block in the CFG can be collected by solving dataflow equations. Taking object-oriented programming as example, on the basis of traditional dataflow analyses, the paper proposes a dataflow analyses method of program with exception-handling constructs. In the analysis process, first we use dataflow graph to represent original program information containing no exception-handling constructs, and then introduce dataflow information of exception-handling constructs.

Exception propagation influences program dataflow information, which mainly includes the following several aspects.

(1) The influence of exception propagation on definition-reach set. Exception propagation may increase or decrease the points where variables definition point can reach.

(2) Exception propagation may increase the active points of some variables.

(3) Exception propagation also has two effects on "definition-use" chain. On the one hand, exception propagation may cause some "definition-use" chains lost; on the other hand, exception propagation may lead to the increase of new "definition-use" chain.

Therefore, on the basis of CFG of program with exception constructs, when analyzing dataflow information of the program, we should consider the influence of exception propagation path on definition-reach set, active variable and "definition-use" chain.

To the problem of dataflow analyses for program with exception constructs, there are mainly two methods. The first is to increase dataflow information of exception-handling constructs in original data flow graph. The second is to express the dataflow information of exception constructs alone. In this paper, we adopt the first one.

Therefore, first we analyze the dataflow information that doesn't take into account exception-handling constructs. As variables definition information, generated definition points of constant definition and type conversion are in definition-reach set OUT_DEF [B].

$$\text{IN\_DEF [B]} = \bigcup_{b \in pred(B)} \text{OUT\_DEF[b]} \tag{1}$$

$$\text{OUT\_DEF [B]} = (\text{IN\_DEF [B]-KILL [B]}) \cup \text{GEN [B]} \tag{2}$$

The following are the sets that are used to compute definition-reach set OUT_DEF [B] [6].

IN_DEF [B]: It represents the definition points set of all variables that can reach basis block B'entrance;

GEN[B]: It represents the definition points set of all variables that are defined in basis block B and can reach B' export;

KILL [B]: It represents the definition points set of all variables that are defined out of basis block B and redefined in B.

OUT_DEF [B]: It represents the definition points set of all variables that can reach basis block B' export.

As variables use information, generated use points of variables in predicates expression are in use-reach set OUT_USE [B]. The following are some sets that are used to calculate use-reach set OUT_USE [B].

(1)GEN[B]: It represents the definition points set of all variables that are defined in basis block B and can reach B' export;

(2)USE [B]: It represents the use points set of all variables that are used in basis block B but not redefined in B after using;

(3)IN_USE [B]: It represents the use points set of all variables that can reach basis block B'entrance;

(4)OUT_USE [B]: It represents the use points set of all variables that can reach basis block B' export.

Where, GEN [B] and USE [B] can be calculated from basis blocks information. IN_USE [B] is the sum of use sets that can reach the exit of all predecessors. IN_USE [B] and OUT_USE [B] can be calculated by (3) and (4).

$$\text{IN\_USE [B]} = \bigcup_{b \in pred(B)} \text{OUT\_USE[b]} \tag{3}$$

$$\text{OUT\_USE [B]} = (\text{IN\_USE [B]-GEN [B]}) \cup \text{USE [B]} \tag{4}$$

When using the information of OUT_USE [B] to analyze the correlations of basis blocks, conditional predicates and use points need to be removed from OUT_USE [B].

3.3. Calculation of definition-reach set and use-reach set for exception variables

In this subsection, we merge exception-handling dataflow information into the original data flow graph. Exception propagation would change certain relationships of variables in a program. In order to obtain precise data flow information, we must consider the influence of exception propagation on the relations of variable definition and use.

The exception variables are those variables that are thrown by *throw* statements and are captured in *catch* statements [7].

Definition point set of all exception variables that arrive at *throw* nodes is denoted t-DEF. To calculate t-DEF, the following sets need to be defined for all of the basis blocks in a program [6].

IN [B]: It represents all the definition points set of exception variables that can reach basis block B'entrance;

OUT [B]: It represents all the definition points set of exception variables that can reach basis block B' exit.

GEN [B]: It represents all the definition points set of exception variables that are defined in basis block B and can reach B' exit.

KILL [B]: It represents the definition points set of exception variables that are defined out of basis block B and have been redefined in B.

IN [B] and OUT [B] need to meet the following formulas:

$$\text{OUT }[B] = (\text{IN }[B]\text{-KILL }[B]) \cup \text{GEN }[B] \tag{5}$$

$$\text{IN }[B] = \cup \text{ OUT }[P] \tag{6}$$

Where, P is the set of all predecessors of basis blocks B.

Usually the linear equations (5) and (6) are solved by iteration method to gain the value of IN [B] and OUT [B]. Iterative algorithm is shown in algorithm 1 of reference [7].

Similarly, use point set of exception variables that arrive at *catch* nodes is denoted c-USE. To calculate c-USE, the following sets need to be defined for all of the basis blocks in a program.

IN [B]: It represents use set of all the exception variables that can reach basis block B'entrance;

OUT [B]: It represents use points set of all the exception variables that reach basis block B' export.

DEF [B]: It represents (s, v) set. Where, 's' uses exception variables v's statement, s is not in B, and B contains v's definition;

USE [B]: It represents (s, v) set. Where, s is the statement that uses exception variables v in B, but v is not defined before B.

Where, DEF [B] and USE [B] can be calculated from exception control flow graph. IN_USE [B] and OUT_USE [B] need to meet formulas (7) and (8):

$$\text{IN\_USE }[B] = \text{USE }[B] \cup (\text{OUT\_USE}[B]\text{-DEF}[B]) \tag{7}$$

$$\text{OUT\_USE }[B] = \cup(\text{IN\_USE}[S]) \tag{8}$$

Where, S is the set of all successors of basis blocks B.

The algorithm of calculating c-USE is shown in algorithm 2 of reference [7].

3.4. Detection of infeasible paths

Conditional predicates are influenced by statements of current basis block or other basis blocks. Therefore, the value of condition predicate is always *true* or *false*. The feature leads to the correlations of different conditional statements. The value of predicate is judged by querying the definition-reach set and use-reach set of basis block where the predicate statements are. The value of predicate includes two types which are *true* or *false*. Particularly, on the analysis of definition point information, variable definition may cause the phenomenon of variable substitution and query alternatives, in this case, we need to query the new replaced variable information in the definition-reach set and use-reach set of basis block where definition point to.

3.4.1. Determination of infeasible branch according to the correlations of conditional statements

According to the definition of correlation, if correlation is generated between two conditional statements Ci and Cj, Ci makes Cj always point to fixed successor. If the value of Cj is always *true*, then its corresponding *false* value branch is infeasible. In contrast, if the value of Cj is always *false*, then its corresponding *true* value branch of Cj is infeasible.

In order to detect infeasible basis paths, our method is to query the definition-reach information and use-reach information of each basis block i (BB[i]). If there existing a definition point of variable v that can make the conditional statements containing v always *true* (*false*), then the corresponding *false* (*true*) value branch of BBi is infeasible.

3.4.2. Detection of infeasible basis paths

The concrete procedure of detecting infeasible basis paths for program with exception constructs is: First, analyze the dataflow information for the program with exception-handling constructs, and generate variable' definition information and use information in each basis block; then, analyze the correlations of different conditional statements according to the definition information and use information; finally, determine the feasibility of branch according to the correlations of conditional statements. The basis paths which contain infeasible branch are considered infeasible.

4. Case study

In this section, we explain and validate the method presented in this paper through an example program with exception-handling constructs. Figure 1 shows its codes. The statements are labelled as (1) ~ (23). Where, the italic codes are the exception-handling constructs.

The corresponding CFG of the sample program P is shown in Figure 2. Where, basis blocks BB [7], BB [10], BB [13], BB [14], BB [15] and BB [16] are exception-handling modules. The branches which are represented with the bold solid line are the added branches with the introduction of exception-handling constructs. Exception propagation makes new "definition-use" lines generated. That is, new paths are generated—exception propagation paths.

```
Main () {
(1)   int n=-1, m, x=2, y, z=1;
(2)   cin>>y;
(3)   if(y<0) m=1;
(4)   else m=-1;
(5)   if (n>0) z=x+1;
(6)   else  {
(7)   try {
(8)    if (z= =0) {
(9)     m=0; throw E1;}
(10)   else
(11)    x=3/z ;}
(12)  catch (exception E1){
(13)     if(x= =2)
(14)       z=2;
(15)     else
(16)       z =0;
(17)     show message E1;}}
(18)   if (x==0)
(19)     z=x-1;
(20)   else
(21)     z=5;
(22)   y=z+1;
(23)  }
```

Figure 1. Codes of Sample Program P

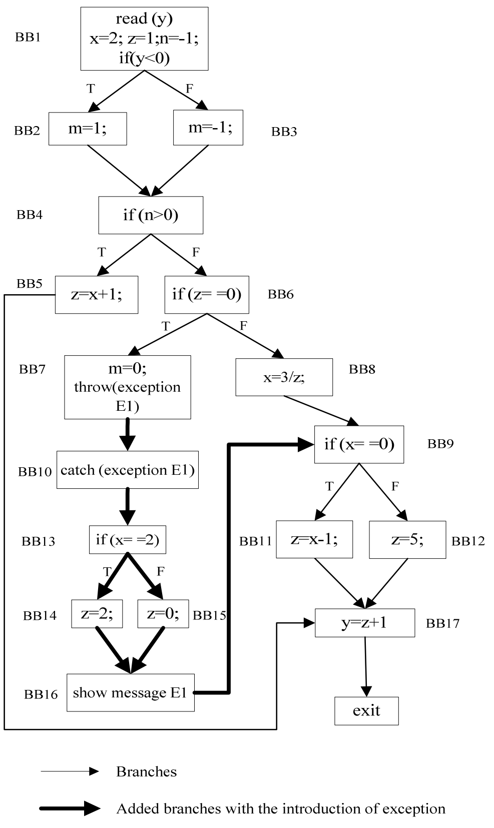

Figure 2. CFG of Sample Program P

In order to analyze the influence of exception-handing constructs on the feasibility of paths, we first ignore the exception-handling constructs. The definition-reach information and use-reach information of each basis block which is calculated by the method of section 3. B is shown in Table 1. (The initial value of In is set 0). When considering exception constructs, the calculating results of def-reach data flow equation is shown in Table 2.

Table 1. Def-reach information and use-reach information of each basis block (without considering exception constructs)

BB[n]	OUT_DEF BB[i]	OUT_USE BB[i]
BB[1]	(1*,x,1) (2*,y,1) (3*,z,1) (4*,n,1)	(5*,y,1)
BB[2]	(1*,x,1) (2*,y,1) (3*,z,1) (4*,n,1) (6*,m,2)	(5*,y,1)
BB[3]	(1*,x,1) (2*,y,1) (3*,z,1) (4*,n,1) (7*,m,3)	(5*,y,1)
BB[4]	(1*,x,1)(2*,y,1)(3*,z,1)(4*,n,1)(6*,m,2) (7*,m,3)	(5*,y,1)(8*,n,4)
BB[5]	(1*,x,1)(2*,y,1)(4*,n,1)(6*,m,2)(7*,m,3) (10*,z,5)	(5*,y,1)(8*,n,4) (9*,x,5)
BB[6]	(1*,x,1)(2*,y,1)(3*,z,1)(4*,n,1)(6*,m,2) (7*,m,3)	(5*,y,1)(8*,n,4) (11*,z,6)
BB[7]	(1*,x,1)(2*,y,1)(3*,z,1)(4*,n,1)(12*,m,7) (13*,E1,7)	(5*,y,1)(8*,n,4)(11*,z,6)
BB[8]	(2*,y,1)(3*,z,1)(4*,n,1)(6*,m,2)(7*,m,3) (15*,x,8)	(5*,y,1)(8*,n,4)(11*,z,6) (14*,z,8)
BB[10]	(1*,x,1)(2*,y,1)(3*,z,1)(4*,n,1)(12*,m,7) (6*,m,2) (7*,m,3) (15*,x,8)	(5*,y,1)(8*,n,4)(11*,z,6) (14*,z,8) (16*,x,9)
BB[12]	(1*,x,1)(2*,y,1) (4*,n,1)(12*,m,7)(6*,m,2) (7*,m,3) (15*,x,8) (18*,z,11)	(5*,y,1)(8*,n,4)(16*,x,9) (17*,x,11)
BB[13]	(1*,x,1)(2*,y,1)(4*,n,1)(12*,m,7)(6*,m,2) (7*,m,3) (15*,x,8) (19*,z,12)	(5*,y,1)(8*,n,4) (16*,x,9)
BB[18]	(1*,x,1)(4*,n,1)(6*,m,2)(7*,m,3)(10*,z,5)(12*,m,7) (15*,x,8)(18*,z,11)(19*,z,12) (21*,y,17)	(9*,x,5)(8*,n,4)(16*,x,9) (17*,x,11) (20*,z,17)

Table 2. Def-reach information and use-reach information of each basis block (consider exception constructs)

BB[n]	OUT_DEF BB[i]	OUT_USE BB[i]
BB[1]	(1*,x,1) (2*,y,1) (3*,z,1) (4*,n,1)	(5*,y,1)
BB[2]	(1*,x,1) (2*,y,1) (3*,z,1) (4*,n,1) (6*,m,2)	(5*,y,1)
BB[3]	(1*,x,1) (2*,y,1) (3*,z,1) (4*,n,1) (7*,m,3)	(5*,y,1)
BB[4]	(1*,x,1)(2*,y,1)(3*,z,1)(4*,n,1)(6*,m,2) (7*,m,3)	(5*,y,1) (8*,n,4)
BB[5]	(1*,x,1)(2*,y,1)(4*,n,1)(6*,m,2)(7*,m,3) (10*,z,5)	(5*,y,1)(8*,n,4) (9*,x,5)
BB[6]	(1*,x,1)(2*,y,1)(3*,z,1)(4*,n,1)(6*,m,2) (7*,m,3)	(5*,y,1)(8*,n,4) (11*,z,6)
BB[7]	(1*,x,1)(2*,y,1)(3*,z,1)(4*,n,1) (12*,m,7)(13*,E1,7)	(5*,y,1)(8*,n,4) (11*,z,6)
BB[8]	(2*,y,1) (3*,z,1)(4*,n,1)(6*,m,2)(7*,m,3) (15*,x, 8)	(5*,y,1)(8*,n,4)(11*,z,6) (14*,z, 8)
BB[9]	(2*,y,1)(3*,z,1)(4*,n,1)(6*,m,2)(7*,m,3)(15*,x,8) (1*,x,1) (12*,m,7)(13*,E1,7) (22*,z,14 (23,z,15)	(5*,y,1)(8*,n,4)(11*,z,6) (14*,z, 8) (16*,x, 9)
BB[10]	(1*,x,1)(2*,y,1)(3*,z,1)(4*,n,1)(12*,m,7) (13*,E1,7)	(5*,y,1)(8*,n,4)(11*,z,6) (17*,E1,10)
BB[11]	(2*,y,1)(4*,n,1)(6*,m,2)(7*,m,3)(14*,x,8) (19*,z,11)	(5*,y,1)(8*,n,4)(16*,x,9) (18*,x,11)
BB[12]	(2*,y,1)(4*,n,1)(6*,m,2)(7*,m,3)(14*,x,8) (20*,z,12)	(5*,y,1)(8*,n,4)(16*,x,9)
BB[13]	(1*,x,1)(2*,y,1)(3*,z,1)(4*,n,1)(12*,m,7)(13*,E1,7)	(5*,y,1)(8*,n,4)(11*,z,6) (17*,E1,10) (21*,x,13)
BB[14]	(1*,x,1)(2*,y,1)(4*,n,1)(12*,m,7)(13*,E1,7) (22*,z,14)	(5*,y,1)(8*,n,4)(17*,E1,10) (21*,x,13)
BB[15]	(1*,x,1)(2*,y,1)(4*,n,1)(12*,m,7)(13*,E1,7) (23*,z,15)	(5*,y,1)(8*,n,4)(17*,E1,10) (21*,x,13)
BB[16]	(1*,x,1)(2*,y,1)(4*,n,1)(12*,m,7)(13*,E1,7) (22*,z,14) (23*,z,15)	(5*,y,1)(8*,n,4)(17*,E1,10) (21*,x,13) (24*,E1,16)
BB[17]	(1*,x,1)(4*,n,1)(6*,m,2)(7*,m,3)(10*,z,5) (14*,x,8)(17*,z,11) (18*,z,12) (26*,y,17)	(8*,n,4)(9*,x,5)(16*,x,9) (18*,x,11) (25*,z,17)

OUT_DEF BB[i] and OUT_USE BB[i] in Table 1 and Table 2 take the form of (a*, b, c). Where, 'a' is the serial numbers, which presents the appearance priorities of all variables' definitions and uses starting with BB [1]. 'b' represents variables in BB[i]. 'c' represents the basis block that the variables are currently in. For example, (4*, n, 1), it represents to define value for variable n in BB [1], and the order of definition is ranked 4.

Then detect the infeasible paths for the sample program. Specific process is as follows:

First, obtain a basis path that does not contain cycle. Path1: 1→2→4→6→7→10→13→14→16→9→12→17.

Then, invert the CFG according to the baseline method, and we can get other 5 paths. All of the independent paths are as following:

Path1: 1→2→4→6→7→10→13→14→16→9→12→17;
Path2: 1→3→4→6→7→10→13→14→16→9→12→17;
Path3: 1→2→4→5→17;
Path4: 1→2→4→6→8→9→12→17;
Path5: 1→2→4→6→7→10→13→15→16→9→12→17;
Path6: 1→2→4→6→7→10→13→14→16→9→11→17.

Finally, detect the feasibility of all the paths in the basis path set. Concrete steps are described in the following:

Step 1, analyze the dataflow information of the program, and calculate def/use reach information of each basis block by the method of section 3.2 (Table 1 and Table 2).

Step 2, find infeasible branches of each basis block according to the def/use reach information of each basis block. Our method is to analyze whether conditional branches statements if (y<0), if (n>0), if (z= =0), if (x= =0) and the branch statement if (x= =2) which is contained in exception-handling statement have branch correlations. Specific process is as follows:

First, analyze definition point or use point of variable y in conditional branch statement if (y<0). Because the statement if (y<0) is in BB [1], remove its own use point of y<0 in OUT_DEF BB [1] and OUT_USE BB [1], leaving (2*, y, 1); then, analyze express read (y) that (2*, y, 1) belongs to, where y is a random number, therefore the value of if (y<0) is undefined. Thus, the two branches of basis block BB [1] may be both are feasible paths.

Then, analyze conditional branch statement if (n>0), which is in BB [4]. Remove its own use point of n>0 in OUT_DEF BB [4] and OUT_USE BB [4], leaving (4*, n, 1). (4*, n, 1) is related to BB[1], and the expression (4*, n, 1) belongs to n=-1, therefore, it makes the value of expression if (n>0) in BB[4] is *false*. Therefore, the *true* branch of BB [5] is infeasible branch.

Next, through analyzing conditional branch statement if (z= =0) in BB [6], we can find that this conditional branch statement causes the execution of exception statements. Similarly, the basis block which is related to BB [6] is BB [1], the express if (z= =0) in BB [6] can be judged according to the statement z= =0 in BB [1], and the result is true. Therefore, the false branch of BB [6] is infeasible branch. Similarly, the value of conditional branch statement if (x= =2) in BB [13] is true. Therefore, the false branch of BB [13] is infeasible branch.

Then, analyze conditional branch statement if (x= =0), which is in BB [9]. We need to query the correlated definition of x in OUT_DEF BB [9], there are two definition which are (15*, x, 8) and (1*, x, 1), but because the false branch of BB [6] is infeasible, the successor which is in the false branch of BB [6] is needed not to be analyzed, so (15*, x, 8) is need to be removed. Therefore, if (x= =0) is judged according to (1*, x, 1), and the result is false, so the true branch of BB [9] is infeasible branch. Thus, the infeasible branches of each conditional statement are obtained.

Step 3, determine the feasibility of paths. The basis paths which contain one or more infeasible branches are considered infeasible. Table 3 shows the results of branches infeasibility.

According to the results of branches feasibility, we can obtain the results of basis paths feasibility for the sample program. It is shown in Table 4.

Table 3. Results of branches feasibility

Condition statements	Basis block where condition branch is in	Infeasible branch of two branches
if(y<0)	BB[1]	undefined
if(n>0)	BB[4]	true branch 4→5
if(z= =0)	BB[6]	false branch 6→8
if(x= =0)	BB[9]	True branch 9→11
if(x= =2)	BB[13]	false branch 13→15

Table 4. Results of basis paths feasibility

No.	Basis paths in basis path set	feasibility
1	1→2→4→6→7→10→13→14→16→9→12→17	feasible
2	1→3→4→6→7→10→13→14→16→9→12→17	feasible
3	1→2→4→5→17	infeasible
4	1→2→4→6→8→9→12→17	infeasible
5	1→2→4→6→7→10→13→15→16→9→12→17	infeasible
6	1→2→4→6→7→10→13→14→16→9→11→17	infeasible

If exception-handling constructs is ignored, with the same analysis method, we can obtain the results of basis paths feasibility for the sample program, which are shown in Table 5.

Table 5. Results of basis paths feasibility

No.	Basis paths in basis path set	feasibility
1	1→2→4→6→8→9→12→17	infeasible
2	1→3→4→6→8→9→12→17	infeasible
3	1→2→4→5→17	infeasible
4	1→2→4→6→7	feasible
5	1→2→4→6→8→9→11→17	infeasible

From the results of basis paths feasibility in Table 4 and Table 5, we can find that exception-handling constructs have influence on paths, which are different from the execution paths of program when ignoring exception-handling constructs. For the sample program, we have detected all infeasible paths.

5. Experiment

To evaluate the effectiveness of the proposed approach we have applied the approach to detect infeasible paths in many programs written in Java. We select some programs that are downloaded from the system 'Software artifact Infrastructure Repository' for experiments [20]. Table 6 shows their detailed descriptions. A disjoint-set is a data structure for maintaining dynamic partitions of a set. A red-black tree is a self balancing tree whose search/insert/remove methods have a time complexity of O (logn). Sorting algorithms take a group of elements and return them in a specific order. A vector is a data structure that implements a mapping from an unbounded range of integers to objects of a certain type. The first column gives the name of each program. Column LOC gives the line of code. Column # basis path means the number of basis paths when exception constructs taken into consideration. Column # basis path' means the number of basis paths when exception constructs are not taken into consideration.

Table 6. Description of target programs

Program	LOC	# basis path	#basis path'
Disjoint-Set	176	20	18
Red-Black-Tree	910	75	70
Sorting	346	31	28
Vector	1227	46	40

In addition, we also implemented a tool to find the basis set of paths in a CFG according to the algorithm in [18].

For each program under test, we first obtain its CFG by SOOT[19]. SOOT is a powerful open-source tool to analyze Java byte code. It provides basic program analysis functions such as control/data flow analysis, which can provide definition/use information. Then, we compute the basis set on the basis of CFG by the tool which is implemented according to the algorithm in [18]. Finally, we invoke the proposed approach to detect the infeasible paths in the basis set, that is, the essence is to detect its infeasible paths according to the correlations of all conditional statements.

Table 7 shows the statistic results of our experiment on the four systems. The first column means the name of the system; I_{total} refers to the number of infeasible basis paths when considering exception constructs; I'_{total} gives the number of infeasible basis paths without considering exception constructs; DI_{total} means the number of infeasible paths detected by our method when considering exception constructs; $\mathbf{DI}'_{total}$ means the number of infeasible paths detected by our method when we considering exception constructs. Re is the ratio of the number of infeasible paths detected by our method to that of infeasible ones of the program when we considering exception constructs.

Table 7. Statistic experimental results

Program	$\mathbf{I}_{total}$	$\mathbf{I'}_{total}$	$\mathbf{DI}_{total}$	$\mathbf{DI'}_{total}$	**Re(%)**
Disjoint-Set	14	13	14	13	100
Red-Black-Tree	61	58	57	56	93.4
Sorting	24	22	23	20	95.8
Vector	37	34	35	32	94.6

We can observe from Table VII that, for the four programs, we have detected almost all of the infeasible paths. But we cannot detect all infeasible paths; the reason is that in this experiment, for the situation that branches feasibility is undefined, the value of conditional statements is decided by the input data, in this case, we consider both of the two branches are feasible. In the experiment, there are while statements, for statements and if statements included in the tested programs. In order to facilitate the experiment and meet to the demand of proposed method in this paper, we convert while and for statements expressions into the form of if statement. In addition, from the experimental results of tested programs, we can see that the influence of exception-handling constructs on the paths. Therefore, infeasible basis paths detection of program with exception-handling constructs is an important part in path-oriented test.

To sum up, our method can detect almost all infeasible paths of some real-world programs; therefore, it is an effective method of detecting infeasible paths of a program.

6. Related works

The existed methods of detecting infeasible paths can be classified into two categories, which are static analysis and dynamic techniques.

In static analysis, the purpose of determining branch correlations is to determine the feasibility of a target path by investigating the branch correlations of different conditional statements. Malevris et al. [9] observed that the more the number of conditional statements contained in a path, the greater the probability of the path being infeasible. It is difficult to determine the branch correlations of different conditional statements exactly.

There are many static approaches to detect infeasible paths. For most of them, the essence is to detect infeasible paths through analyzing the correlations of different conditional statements.

Bodik et al. [2] and Chen et al. [8] have proposed their own algorithms for detecting infeasible paths incorporating the branch correlations analysis into the dataflow analyses technique, which improve the precision of traditional dataflow analyses. Chen et al. [10] proposed a method of detecting infeasible paths by investigating whether there are conflicts between the assignment statement and the branch statement, as well as among different branch statements. Ngo et al. [3] also proposed a heuristics-based approach to detect infeasible paths, which is based on the observation that many infeasible paths exhibit some common properties which are caused by four code patterns. Through realizing these properties, many infeasible paths can be precisely detected but with some limitations in the current prototype tool. Gong et al. [11] proposed an approach to automatic detection of infeasible paths. They determined branch correlations based on the probabilities of the conditional distribution corresponding to different branches' outcome. The values of these probabilities are obtained by the maximum likelihood estimation. Delahaye et al. [21] proposed a new method that takes opportunity of the detection of a single infeasible path to generalize to a (possibly infinite) family of infeasible paths, which will not have to be considered in further path conditions solving. Another static technique of detecting infeasible paths relies on the symbolic execution [12, 13]. However, the approach is expensive; besides, a large percentage of infeasible paths can not be detected for a complicated program with arrays, pointers as well as procedure calls.

The above static analysis approaches all ignored the exception-handling constructs.

For dynamic approaches to detect infeasible paths, dynamic test data generation algorithms can be used to detect infeasible paths by monitoring the execution of a program [14, 15]. Ngo et al. [3] proposed a heuristics-based approach to detect infeasible paths, which is based on the characteristic that many infeasible paths exhibit some common properties. During test data generation, through distinguishing the properties in executing traces, infeasible paths can be detected. Moreover, Balakrishnan et al. [16] presented a syntactic language refinement technique that automatically excludes semantically infeasible paths from a program. Ngo et al. [17] have proposed an approach to detect infeasible paths during dynamic test data generation process.

However, test data generation often relies on symbolic execution, so dynamic approaches are almost as expensive as symbolic execution.

In this paper, we adopt a new static analysis approach to detect infeasible paths of programs with exception-handling constructs. The technique combines exception propagation with the branch correlations of different conditional statements to analyze the impacts of exception propagation on paths. The correlations are determined according to dataflow analyses.

7. Conclusions

We have proposed an approach to detect infeasible basis paths based on the correlations of different conditional statements. The correlations are determined according to dataflow analyses. In order to attain more precise dataflow information, as well as to further improve the accuracy of the feasibility of paths, we take account of exception propagation and during the analyses. Then, we apply the approach of detecting infeasible basis paths to a case study in detail. Finally, we have also conducted experiments to evaluate the effectiveness of the proposed approach. Even with some limitations in the current prototype tool, the experimental results show that our approach can accurately detect infeasible paths.

Pointer alias analysis plays an important role on analysis techniques in C/C+ + program, such as data flow analysis, program optimization and program environment and analysis tools. It is an effective way to get more accurate pointer information, improve precision of other data flow analysis and correlation analysis. Currently, we are performing experiments on the program with pointer alias to detect infeasible program paths accurately.

8. Acknowledgment

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

9. References

[1] David Hedley, Michael A. Hennell, "The Causes and Effects of Infeasible Paths in Computer Programs", In Proceedings of the 8th International Conference on Software Engineering, pp. 28-30, 1985.

[2] Rastislav Bodík, Rajiv Gupta, Mary Lou Soffa, "Refining Data Flow Information Using Infeasible Paths", In Proceedings of the 6th Europe Software Engineering Conference and 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 361-377, 1997.

[3] Minh Ngoc Ngo, Hee Beng Kuan Tan, "Detecting Large Number of Infeasible Paths through Recognizing Their Patterns", In Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering, pp. 215-224, 2007.

[4] Thomas J. McCabe, "Structural Testing: A Testing Methodology Using the Cylomatic Complexity Metric", NIST Special Publication 500-99, Washigton D.C, 1982.

[5] Jun Yan and Jian Zhang, "An Efficient Method to Generate Feasible Paths for Basis Path Testing", Information Processing Letters, vol. 107, pp. 87-92, 2008.

[6] Liyuan Jiang, Muning Kang, "Compilers Principles", Xi'an: Northwest Industrial University press, China, 2006.

[7] Shujuan Jiang, Baowen Xu, Liang Shi, Xiaoyu Zhou, "An Approach to Analyzing Dependence based on Exception Propagation analysis", Chinese Journal of Software, vol. 18, no.4, pp.832-841, 2007.

[8] Rui Chen, "Infeasible Path Identification and Its Application in Structural Test", Beijing: Graduate School of Chinese Academy of Science, 2006.

[9] Nicos Malevris, "A Path Generation Method for Testing LCSAJs that Restrains Infeasible Paths", Information Software Technology, vol. 37, pp. 435-441, 1995.

[10] Ting Chen, Tulika Mitra, Abhik Roychoudhury, Vivy Suhendra, "Exploiting Branch Constraints without Exhaustive Path Enumeration", In Proceedings of the 5th International Workshop on Worst-Case Execution Time Analysis, pp. 40-43, 2005.

[11] Dunwei Gong, Xiangjuan Yao, "Automatic Detection of Infeasible Paths in Software Testing", Journal of IET software, vol. 4, no.5, pp.361-370, 2010.

[12] Antonella Santone, Gigliola Vaglini, "Formula-based Abstractions and Symbolic Execution for Model Checking Programs", Microprocessors and Microsystems, vol. 28, pp. 69-76, 2004.

[13] Bahman Pourvatan, Marjan Sirjani, Hossein Hojjat, Farhad Arbab, "Automated Analysis of Reo circuits Using Symbolic Execution", Electronic Notes in Theoretical Computer Science, vol. 255, pp. 137-158, 2009.

[14] Neelam Gupta, Aditya P. Mathur, Mary Lou Soffa, "Generating Test Data for Branch Coverage", In Proceedings of ASE 2000 15th IEEE International Automated Software Engineering Conference, pp.11-15, 2000.

[15] Paulo Marcos Siqueira Bueno, Mario Jino, "Identification of Potentially Infeasible Program Paths by Monitoring the Search for Test Data", In Proceedings of the fifteenth IEEE International Automated Software Engineering Conference, pp. 209-218, 2000.

[16] Gogul Balakrishnan, Sriram Sankaranarayanan, Franjo Ivančić, Ou Wei, "SLR: Path-Sensitive Analysis through Infeasible-Path Detection and Syntactic Language Refinement", (LNCS, 5079), pp.238-254, 2008.

[17] Minh Ngoc Ngo, Hee Beng Kuan Tan, "Heuristics-based Infeasible Path Detection for Dynamic Test Data Generation", Information and Software Technology, vol. 50, pp. 641-655, 2008.

[18] Joseph Poole, "A Method to Determine a Basis Set of Paths to Perform Program Testing", United States, 1995. (compressed Postscript, 68k)

[19] SOOT, "A Java Bytecode Optimization Framework", http://www.sable.mcgill.ca/soot/.

[20] SIR, "A Repository of Software-Related Artifacts Meant to Support Rigorous Controlled Experimentation", Available at: http://sir.unl.edu/portal/index.html.

[21] Mickaël Delahaye, Bernard Botella, Arnaud Gotlieb, "Explanation-based Generalization of Infeasible Path", In Proceedings of the 3rd International Conference on Software Testing, Verification and Validation, pp.215-224, 2010.

[22] Rachida Saouli, Mohamed Akil, Thierry Grandpierre,"Learning System for Defactorization Factor Classification of Factorized Data Dependence Graph", IJACT: International Journal of Advancements in Computing Technology, vol. 3, no. 4, pp. 1-13, 2011.

[23] Osamu Mizuno, Hideaki Hata, "A Hybrid Fault-Proneness Detection Approach Using Text Filtering and Static Code Analysis", International Journal of Advancements in Computing Technology, vol. 2, no. 5, pp. 1-12, 2010.