IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received April 22, 2019, accepted May 12, 2019, date of publication May 29, 2019, date of current version June 7, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2918558

An Approach for Detecting Infeasible Paths Based on a SMT Solver

SHUJUAN JIANG[1,2], HONGYANG WANG[1,2], YANMEI ZHANG[1,2,3], MENG XUE[1,2], JUNYAN QIAN[3], AND MIAO ZHANG[1,2]

[1]School of Computer Science and Technology, China University of Mining and Technology, Xuzhou 221116, China
[2]Mine Digitization Engineering Research Center of the Ministry of Education, China University of Mining and Technology, Xuzhou 221116, China
[3]Guangxi Key Laboratory of Trusted Software, Guilin University of Electronic Technology, Guilin 541004, China

Corresponding authors: Yanmei Zhang (ymzhang@cumt.edu.cn) and Junyan Qian (qjy2000@guet.edu.cn)

This work was supported in part by the Fundamental Research Funds for the Central Universities under Grant 2017XKZD03.

ABSTRACT Software testing is an important means to ensure software quality. Testers need to ensure that every component of the software is tested correctly to achieve high coverage, such as path coverage, decision coverage, and branch coverage. An infeasible path is a path that cannot be traversed by any test cases. The existence of infeasible paths can waste test resources; therefore, detection of infeasible paths are necessary before path testing. This paper presents a static method for the detecting infeasible paths that is based on a satisfiability modulo theory (SMT) solver. First, the proposed method generates a sub-path set and converts the feasibility issues into inequalities. Second, a constraint solver is used to solve the inequalities and, then, the sub-paths are divided into two categories: infeasible sub-paths and undetermined sub-paths. The paths that were expanded from the latter will be tested again to determine their feasibility. Finally, the feasibility of all paths is detected. Most of the detection works are done on the sub-path set; therefore, our method provides an effective solution to the path-explosion problem. The experimental results showed that the proposed method can detect infeasible paths more accurately and effectively than most existing methods.

INDEX TERMS Software testing, software quality, sub-path expansion, infeasible path detection, constraint solving.

I. INTRODUCTION

The existence of infeasible paths has major impact to many software engineering activities. The code can certainly be optimized further if more infeasible paths can be detected during the process of optimization. During the process of software testing, if the test data is for those statements which are in infeasible paths, then the data will not actually be tested, which would cause much waste. Therefore, in software testing, the structural test coverage can be much accurately computed if infeasible paths can be detected more accurately. During the process of test case generation, much time can be saved if more infeasible paths can be detected, which can reduce the waste of resources. In code protection, it can also help to identify the inserted spurious paths in code deobfuscation. In software verification, detecting and eliminating infeasible paths will help to enhance the verification precision and speed [1]. Therefore, detection of these infeasible paths has a key impact on many software engineering activities including code optimization, testing and even software security [1].

Infeasible path detection methods can be divided into three categories, static methods, dynamic methods, and hybrid methods of combining static methods with dynamic methods. For the static methods, they are based on the branch correlation analysis method or on the satisfiability of the path condition method. However, the branch correlation analysis method has low accuracy and the satisfiability of the path condition method cannot deal with the path-explosion problem for large-scale programs. For the dynamic methods, they use a heuristic search algorithm to search for test cases to cover the target path. If the test cases cannot be found at a certain search depth, the path can be deduced to be infeasible. The search depth of such methods needs to be modified according to the size of the programs. Therefore, such methods have very poor adaptability. For the hybrid methods, they use the information from the static analysis such as branch correlation analysis or path condition analysis to improve the dynamic heuristic search process. Although a static analysis can assist the

The associate editor coordinating the review of this manuscript and approving it for publication was Xiaobing Sun.

heuristic search process, there are still some shortcomings such as a large overhead, path-explosion problems, etc.

To illustrate the research motivation, we use the example program in Figure 1.

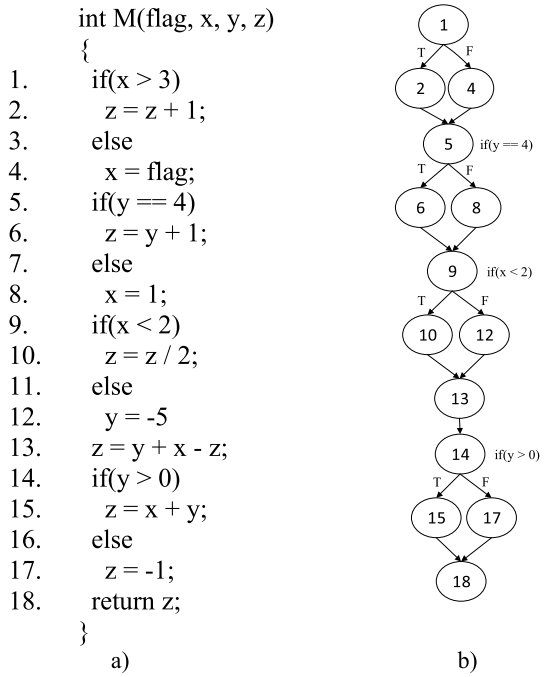

FIGURE 1. An example program and its CFG. (a) An example program. (b) CFG.

Figure 1 (a) is the source code of the example program and Figure 1 (b) is its CFG. This program has many infeasible paths, such as sp_1:1,2,5,6,9,10,13,14,17,18; sp_2:1,4,5,6,9,12,13,14,15,18. For sp_1, the feasibility requires that we must satisfy the two conditions: (x $>$ 3, y $=$ 0) and (x $<$ 2, y $<=$ 0). Obviously, the two condition is conflicting. For sp_2, the feasibility requires that we must satisfy the two conditions: (y $= -5$) and (y $= 0$). Obviously, the two conditions are also conflicting.

First, it would be a major waste when generating test case if not detected the infeasible paths. Second, it would cause path explosion if there are a lot of branch nodes or cyclic nodes.

On the basis of the state-of–the-art research level and the existing problems, this paper presents a static method based on a satisfiability modulo theory (SMT) solver named SMT-IPD (SMT solver based on Infeasible Path Detection) to detect infeasible paths. First, we generate a sub-path set, from which we obtain the path conditions. Second, we use the constraints in the path condition to constitute a system of inequalities. Then, an SMT solver is used to solve the system of inequalities. According to the results solved by the SMT solver, we divide the sub-paths into two groups. The first group is the infeasible sub-path group, which can be expanded into infeasible paths. The second group is the feasible or unidentified sub-path group, which can be expanded into the paths that need retesting. Finally, we can get all the paths and their feasibility information.

The main detection work of SMT-IPD is on the sub-path set. This method can solve the path-explosion problem partly because the number of sub-paths is far less than the number of paths for complex programs.

The contributions of this paper are as follows:

We provide an infeasible path detection method with higher accuracy.

We determine the feasibility for each sub-path, and expand the sub-paths into full paths and obtain the feasibility of the full paths.

II. BACKGROUND

In this section, we introduce some concepts used in this paper.

Definition 1 (Path [2]): A path π through a control flow graph CFG G = (V, E, s, x) is a sequence of nodes from the start block to the exit block $\pi = (s, v_1, \ldots, v_n, x)$ with s, $v_1 \ldots v_n$, x $\in$ V and (s, v_1), (v_1, v_2), $\ldots$, (v_n, x) $\in$ E.

Definition 2 (Post-Dominance [3]): Given a CFG G= (V, E, s, x), $\ldots$ a node *m* is post-dominated by a node *n* ($m, n \in$ G) if every valid path from *m* to the end node x contains *n*.

Definition 3 (Predominator Relationships [4]): For nodes n_i and n_j in CFG, if all paths from entrance node s to n_j pass through n_i, it is called n_i predominate n_j, denoted as $n_i \xrightarrow{pre} n_j$. If $n_i \xrightarrow{pre} n_j$, and all other dominator of n_j are n_k' dominator, it is called that n_k directly dominate n_j, denoted as $n_k = idom(n_j)$.

A node predominates itself but does not post-dominate itself.

Definition 4 (Predominator Tree [4]): In a CFG, any other node has a direct **dominator** except node s. According to the dominator relationships, we can construct a Predominator Tree with s as the root node. Predominator Tree can be expressed as a triple (N, E, s), where, E = {(idom(n_i), n_i)|$n_i \in$ N $-$ {s}}.

Definition 5 (Control Dependence [3]): Let G be a control flow graph. Let X and Y be nodes in G. Y is control dependent on X iff

(1) There exists a directed path P from X to Y with any Z in P (excluding X and Y) post-dominated by Y and
(2) X is not post-dominated by Y.

If Y is control dependent on X then X must have two exits.

Definition 6 (Sub-Path (sp)): A sub-path is a segment of a path. The difference between a path and a sub-path is the starting point and the end point. In a path, the starting point is the entry node of the control flow graph (CFG), whereas the end point is the exit node of the CFG. In contrast, in a sub-path, the starting point is the entry node or a branch node of the CFG, whereas the end point is the exit node of the CFG.

There is an assumption that control flow is static. That is, there are no indirect jumps and the CFG can be constructed directly from program text.

Definition 7 (Correlation Variable (Correlation Constants)): Generally, predicate P of a branch statement in a program can be expressed with two forms, v_1 op_1 v_2 and v_3 op_2 c. Where, v_1, v_2 and v_3 are variables, c is a constant,

op_1 and op_2 represent decision symbol (such as greater-than sign, >), then v_1, v_2 and v_3 are called correlation variables of predicate P (c is the correlation constant of predicate P).

Definition 8 (Variable Definition Point): The assignment statement that defines the value of variable v is a variable definition point. If variable v is a correlation variable, it is a correlation variable definition point, denoted by $dp(v)$.

Definition 9 (Infeasible Sub-Path/Path): An infeasible sub-path/path is a sub-path/path that cannot be traversed by any program input.

Definition 10 (Basic Block): A basic block is the maximum sequence of statements with a single entry point and a single exit point.

III. OUR APPROACH: SMT-IPD

This paper presents a static method (SMT-IPD) for detecting infeasible paths based on an SMT solver. Figure 2 shows the framework of SMT-IPD.

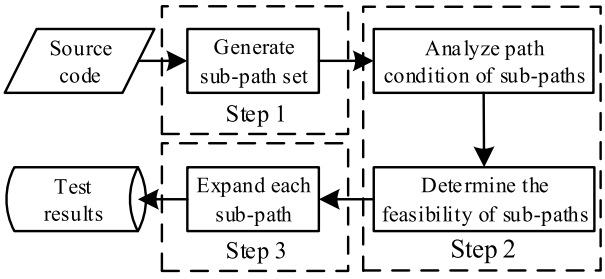

FIGURE 2. Framework of the SMT-IPD.

SMT-IPD consists of three steps.

In Step 1, it generates a sub-path set. Before obtained the sub-path set, it needs to statically analyze the .java or .class file of the Java programs. First, it obtains the CFG of the tested program using the Java program analysis framework Soot,[1] and eliminates the cycle of the CFG to obtain the directed acyclic CFG. Then it calculates the control dependency of the tested program according to the control dependency calculation method. Finally, it recursively traverses the acyclic CFG obtained from the static analysis part to generate a sub-path set according to the sub-path set construction algorithm.

In Step 2, it determines the feasibility for each sub-path. First, it analyses the path condition for each sub-path. Then, it constructs the inequalities for each sub-path according to the inequalities construction algorithm; finally, the inequalities is solved by an SMT solver, and the results show the feasibility of each sub-path. If there is a solution, it means that the sub-path is feasible; otherwise, the sub-path is infeasible or undetermined. For the undetermined paths, it need be tested again to determine their feasibility.

In Step 3, SMT-IPD expands the sub-paths into paths and, finally, obtains the feasibility of all the paths.

[1]https://sable.github.io/soot/

A. GENERATION OF A SUB-PATH SET

For CFGs with loops, we need to add or delete edges to break the loops. A CFG can be treated as a directed acyclic graph and can be traversed recursively to generate a sub-path set.

Breaking loops in a CFG means cutting off the loops in a program. For the loops in a program, we treat them as follows:

1) The code in a loop executes only once. If the loop contains branch statements, the **true** branch and the **false** branch execute only once separately.
2) The code in a loop is not executed. In other words, the code only executes the entry of the loop and then jumps to the exit of the loop without executing the loop body.

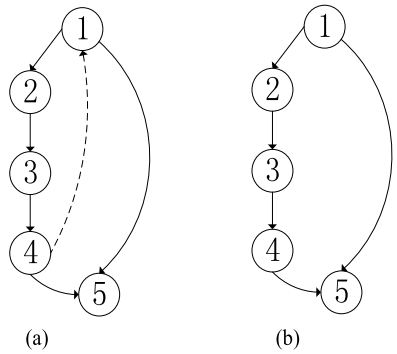

FIGURE 3. An example of breaking the loop in a CFG. (a) Before breaking the loop. (b) After breaking the loop.

Figure 3 shows an example of breaking a loop in a CFG. Figure 3 (a) shows the original CFG, in which 1-2-3-4-1 is a loop. We deleted the dashed edge $4 \rightarrow 1$, which is a backward edge of a loop. This is because the loop can be broken but the other paths would not be changed if the backward edge $4 \rightarrow 1$ was deleted. After deleting edge $4 \rightarrow 1$, we can obtain Figure 3 (b), which is the CFG after breaking the loop.

The CFG contains two paths after breaking the loop.

1) {1, 2, 3, 4, 5}: The code in the loop executes once.
2) {1, 5}: The code in the loop is not executed, which jumps from the entry of the loop to the exit of the loop.

After breaking a loop, SMT-IPD generates sub-path sets according to Algorithm 1.

SMT-IPD generates sub-path sets recursively. Once a loop has been broken, the recursive algorithm could now reach an end point. An end point denotes a certain node in a CFG that has no successors, which means that the condition in line 2 evaluates to **false**. The nodes with no successors are the exit nodes in a CFG. When an end point is reached, *sp* is a sub-path (*sp* will be modified in the recursive process, and, therefore, it is not initialised in line 1). All the nodes in *sp* will be deleted (line 8) after *sp* is added to subPathSet (line 7). Then, according to the call stack of the recursive algorithm, our method gets to the upper layer and explores new sub-paths (line 4).

Algorithm 1 Sub-Path Set Generation

input: startLine //The first line of the program
output: subPathSet
Function subPathGeneration(startLine) //Recursive invocation
Begin
1. sp←sp ∪ startLine
2. **if** startLine has successors **then**
3. **foreach** line in startLine.successors
4. subPathGeneration(line) //Recursive invocation
5. **endfor**
6. **else**
7. subPathSet←subPathSet ∪ sp
8. sp←Ø
9. **endif**

End

```
1.    if x > 3
2.      statement 1
…     …
7.    if x < 2
8.      statement 2
…     …
10.   …y = -5
…     …
15.   if y > 0
16.     statement 3
…     …
```

FIGURE 4. An example of infeasible sub-paths.

B. DETERMINATION OF THE FEASIBILITY FOR EACH SUB-PATH

First, we explain the principle of how to determine the feasibility of sub-paths. Figure 4 shows an example, which has two cases that cause the infeasibility of sub-paths.

1) The conditions of some statements from two branch statements have conflicts. Suppose we have sub-path $sp_1 = \{1, 2, \ldots 7, 8, \ldots\}$, where 1→2 and 7→8 are two branch statements. The branch 1→2 indicates $x > 3$, and the branch 7→8 indicates $x < 2$. Variable x is not defined again between the two branch statements in sub-path sp_1; hence, sub-path sp_1 is infeasible. We denote this case as a B-B (Branch-Branch) conflict.

2) The condition of some statements from branch statements has conflicts with a correlation variable *definition* point. Suppose we have sub-path $sp_2 = \{\ldots 10, 15, 16, \ldots\}$, where 15→16 is a branch and correlation variable y is defined in line 10. The branch 15→16 indicates $y > 0$, but correlation variable y is defined as -5 in line 10 and it is not defined again between line 11 and line 14; thus, sub-path sp_2 is infeasible. We denote this case as an A-B (Assignment-Branch) conflict.

In this step, our method abstracts sub-paths and analyses branch conditions and correlation variable *definition* points. The path conditions are represented as a system of inequalities. Then, an SMT solver is used to solve the system of inequalities to check whether the two cases mentioned above exist or not. If the system of inequalities is satisfied, the sub-path is feasible; otherwise, it is infeasible. If the result of the system of inequalities cannot be determined, the feasibility of the sub-path cannot be identified either, which needs to be checked again.

According to the CFG, our method can obtain the control dependence information between statements. The control dependence information is represented as a predominator tree. The successor nodes in a predominator tree are control dependent on direct and indirect predecessor nodes. After the control dependence information is calculated, SMT-IPD constitutes a system of inequalities for each sub-path based on Algorithm 2.

Algorithm 2 Inequality Set Generation

input: sp //Sub-path
PredominatorTree // predominator tree
output: inequalitySet
Begin
1. inequalitySet←n
2. sp←sp.reverse
3. **foreach** node **in** sp
4. **if** node *is in PredominatorTree*
5. && node *has more than 1 successor* **then**
6. inequalitySet←inequalitySet ∪ node.getBranchPredicate
7. variableSet←variableSet ∪ node.getVariable
8. **else if** node *is a define statement*
9. && node.getVariable ∈ variableSet
10. &&node n *is first met*
11. inequalitySet←inequalitySet ∪ node.getDefineExpression
12. **endif**
13. **endfor**

End

The inequality set generation algorithm traverses every node in each sub-path and checks whether it is a branch node or a correlation variable *definition* node (lines 2-3). If node m is in a predominator tree and has more than 1 successor (lines 4-7), the algorithm extracts predicate p from branch node m and then adds predicate $p$to the system of inequalities. If predicate p contains correlation variable x, the algorithm puts this variable into variableSet.

If node n is a variable *definition* node, in which the variable is a correlation variable, and it is the first node met in the reverse traversal (lines 8-10), the algorithm adds the *definition* of the correlation variable to inequalitySet (line 11).

C. EXPANSION OF SUB-PATHS

In this section, SMT-IPD expands the sub-path set and improves the detection results of infeasible paths. First, SMT-IPD expands each sub-path into a path according to the sub-path set expansion algorithm; then, it determines the feasibility of the expanded paths and completes the detection results.

Algorithm 3 Sub-Path Set Expansion

```
Declare: sp,sp': sub-path
fp: full-path
input: subPathSet      //Sub-path set
       CFGHeadSet    //Heads of the CFG
output: infeasiblePathSet
       unknownPathSet
Begin
1.      infeasiblePathSet←n
2.      unknownPathSet←n
3.      foreach sp in subPathSet
4.          head←sp.head
5.        foreach sp' in subPathSet
6.          if sp' contains head && sp' head ∈ CFGHeadSet
            then
7.            fp←sp'.subList(0, head) ∪ sp
8.            if sp is infeasible || sp' is infeasible then
9.              infeasiblePathSet←infeasiblePathSet ∪ fp
10.           else
11.             unknownPathSet←unknownPathSet ∪ fp
12.           endif
13.         endif
14.       endfor
15.     endfor
End
```

The algorithm first selects a head node for each sub-path sp (lines 3 and 4). Then, it focuses on the other sub-path sp′. If sp′ contains the head node and the first node of sp′ is an entry node of the CFG, then sp′ and sp can be connected to a single path fp (lines 6-7). The algorithm takes a partial sequence from the first node to the head node in sp′ and combines the partial sequence with sp to fp. In other words, fp is a combination of sp and sp′, and the connection point is the common node head of sp. Finally, according to the feasibility of sub-path sp and sp′, path fp falls into the corresponding result set. If sp or sp′ is infeasible, the expanded path fp is also infeasible; if sp and sp′ cannot be determined, the algorithm cannot determine the feasibility of fp directly; therefore, it needs a secondary determination (lines 8-12).

D. AN EXAMPLE

In this section, we use the example program of Figure 1 to illustrate how SMT-IPD works.

Step 1: SMT-IPD generates a sub-path set. subPathSet = $\{sp_1, sp_2, sp_3, sp_4, sp_5\}$.

$sp_1 = (14, 17, 18)$;
$sp_2 = (9, 12, 13, 14, 15, 18)$;
$sp_3 = (5, 6, 9, 10, 13, 14, 17, 18)$;
$sp_4 = (1, 2, 5, 8, 9, 12, 13, 14, 15, 18)$;
$sp_5 = (1, 4, 5, 6, 9, 10, 13, 14, 15, 18)$.

Step 2: SMT-IPD generates inequalities for each sub-path. InequalitySet = $\{(1, x > 3), (4, x = flag), (5, y = 4), (8, x = 1), (9, x < 2), (12, y = -5), (14, y > 0)\}$.

We can get the following several systems of inequalities:

$$sp_1: \quad y \le 0 \rightarrow \text{solvable}$$

$$sp_2: \begin{cases} x \ge 2 \\ y = -5 \\ y > 0 \end{cases} \rightarrow \text{unsolvable}$$

$$sp_3: \begin{cases} y = 4 \\ x < 2 \\ y \le 0 \end{cases} \rightarrow \text{unsolvable}$$

$$sp_4: \begin{cases} x > 3 \\ y \ne 4 \\ x = 1 \\ x \ge 2 \\ y = -5 \\ y > 0 \end{cases} \rightarrow \text{unsolvable}$$

$$sp_5: \begin{cases} x \le 3 \\ x = 0 \\ y = 4 \\ x < 2 \\ y > 0 \end{cases} \rightarrow \text{solvable}$$

The paths expanded from sp_2, sp_3 and sp_4 are infeasible. The paths expanded from sp_1 and sp_5 need to be tested again.

Step 3: SMT-IPD expands each sub-path. Take sub-path sp_3 as an example. The head node of sub-path sp_3 is node 5. Sub-path sp_4 contains node 5, and the head node of sp_4 is one of the entry nodes of the CFG (the head node of the CFG is node 1). Therefore, sp_3 and sp_4 can be connected to a path fp_1. We take sequence $\{1, 2, 5\}$ from sp_4 and combine it with sp_3, and then we can get $fp_1 = \{1, 2, 5, 6, 9, 10, 13, 14, 17, 18\}$.

All the sub-paths from sp_1 to sp_5 can be expanded into 16 paths that contain 10 infeasible paths, as shown at the top of the next page. We use fp_i to represent the expanded path.

Path fp_1 is expanded from infeasible sub-path sp_3. Paths fp_2 and fp_3 are expanded from infeasible sub-path sp_2. Paths fp_4, fp_5, fp_6 and fp_7 are expanded from infeasible sub-path sp_4. These seven paths are infeasible, which can be deduced from infeasible sub-paths sp_2, sp_3 and sp_4. Sub-path sp_5 corresponds to path fp_8, and the feasibility of fp_8 can be determined according to the feasibility of sub-path sp_5. Therefore, SMT-IPD only needs to detect 5 sub-paths ($sp_1, sp_2, \ldots, sp_5$) and 8 expanded paths ($fp_9, fp_{10}, \ldots, fp_{16}$) to obtain the feasibility of all 16 paths. In this example, the number of paths detected by our method was slightly less than the number of all the paths. The complexity of sub-paths is much lower than that of a full path; therefore, it is much easier to detect the feasibility of a sub-path. Therefore, our method is effective in terms of reducing the detection cost.

IV. EMPIRICAL STUDY OF OUR APPROACH

To evaluate our approach empirically, we implemented an infeasible path-detection tool for Java programs and conducted an empirical study using open-source projects.

$$sp_4 \,\&\, sp_3 \Rightarrow fp_1: 1,2,5,6,9,10,13,14,17,18 \rightarrow \begin{cases} x > 3, y = 4 \\ x < 2, y \leq 0 \end{cases} \quad \text{infeasible path}$$

$$\left.\begin{array}{l} sp_5 \,\&\, sp_2 \Rightarrow fp_2: 1,4,5,6,9,12,13,14,15,18 \\ sp_4 \,\&\, sp_3 \,\&\, sp_2 \Rightarrow fp_3: 1,2,5,6,9,12,13,14,15,18 \end{array}\right\} \rightarrow \begin{cases} y = -5 \\ y > 0 \end{cases} \quad \text{infeasible path}$$

$$\left.\begin{array}{l} sp_4 \Rightarrow fp_4: 1,2,5,8,9,12,13,14,15,18 \\ sp_5 \,\&\, sp_4 \Rightarrow fp_5: 1,4,5,8,9,12,13,14,15,18 \end{array}\right\} \rightarrow \begin{cases} x \geq 2, y > 0 \\ x = 1, y = -5 \end{cases} \quad \text{infeasible path}$$

$$\left.\begin{array}{l} sp_4 \,\&\, sp_1 \Rightarrow fp_6: 1,2,5,8,9,12,13,14,17,18 \\ sp_5 \,\&\, sp_4 \,\&\, sp_1 \Rightarrow fp_7: 1,4,5,8,9,12,13,14,17,18 \end{array}\right\} \rightarrow \begin{cases} x \geq 2 \\ x = 1 \end{cases} \quad \text{infeasible path}$$

$$sp_5 \Rightarrow fp_8: 1,4,5,6,9,10,13,14,15,18 \rightarrow \begin{cases} x = 0, y = 4 \\ x < 2, y > 0 \end{cases} \quad \text{feasible path}$$

$$check\ again \Rightarrow fp_9: 1,2,5,6,9,10,13,14,15,18 \rightarrow \begin{cases} x > 3 \\ x < 2 \end{cases} \quad \text{infeasible path}$$

$$check\ again \Rightarrow fp_{10}: 1,4,5,6,9,10,13,14,17,18 \rightarrow \begin{cases} y = 4 \\ y \leq 0 \end{cases} \quad \text{infeasible path}$$

$$check\ again \Rightarrow fp_{11}: 1,2,5,6,9,12,13,14,17,18 \rightarrow \begin{cases} y = -5, x > 3 \\ y \leq 0, x \geq 2 \end{cases} \quad \text{feasible path}$$

$$check\ again \Rightarrow fp_{12}: 1,2,5,8,9,10,13,14,15,18 \rightarrow \begin{cases} x = 1, y \neq 4 \\ x < 2, y > 0 \end{cases} \quad \text{feasible path}$$

$$check\ again \Rightarrow fp_{13}: 1,2,5,8,9,10,13,14,17,18 \rightarrow \begin{cases} x = 1, y \neq 4 \\ x < 2, y \leq 0 \end{cases} \quad \text{feasible path}$$

$$check\ again \Rightarrow fp_{14}: 1,4,5,6,9,12,13,14,17,18 \rightarrow \begin{cases} x = 0, y = -5 \\ x \geq 2, y \leq 0 \end{cases} \quad \text{infeasible path}$$

$$check\ again \Rightarrow fp_{15}: 1,4,5,8,9,10,13,14,15,18 \rightarrow \begin{cases} x = 1, y \neq 4 \\ x < 2, y > 0 \end{cases} \quad \text{feasible path}$$

$$check\ again \Rightarrow fp_{16}: 1,4,5,8,9,10,13,14,17,18 \rightarrow \begin{cases} x = 1, y \neq 4 \\ x < 2, y \leq 0 \end{cases} \quad \text{feasible path}$$

This section describes our experimental setup and presents the empirical results.

A. EXPERIMENTAL SETUP

We first obtained the CFG by using the Java program analysis framework Soot.[2] According to the sub-path set generation algorithm, we generated a sub-path set. Then, we analyzed the predicates of branch statements and the *definition* points of the correlation variables to construct the system of inequalities. We determined the feasibility of sub-paths by solving the system of inequalities, which was completed by SMTInterpol.[3] Finally, we expanded the sub-paths into full paths to obtain the feasibility of all paths. The result was stored in a Sqlite[4] database so we could query the results by using SQL statements. All the experiments were carried out on a Dell server with 32 GB of memory and two 3.07-GHz XEON X5675 CPUs using JDK 1.7. We ran each analysis with about 28 GB of heap memory for the JVM (java -Xmx28000M).

B. SUBJECTS OF THE EXPERIMENTS

To verify the accuracy and the effectiveness of our method, we designed three groups of experiments, where the subjects in the first group of experiments were benchmark programs and the subjects in the second group and the third group of experiments were from the SIR[5] website. The information is shown in Table 1. The table shows, for each subject, the group number, program name (columns 1 and 2), description, lines

[2]https://sable.github.io/soot/
[3]http://ultimate.informatik.uni-freiburg.de/smtinterpol/
[4]http://www.sqlite.org/

[5]http://sir.unl.edu/php/previewfiles.php?lang%5B%5D=Java&name=&min_ver_cnt=&max_ver_cnt=&min_src_size=&max_src_size=&min_unit_cnt=&max_unit_cnt=&display=Display

TABLE 1. Subjects of the experiments.

Group	Program	Description	Lines of code	Number of Classes	Number of Methods
1	Zip viewer	Zip tool	65	2	5
	Bubble sort	Bubble sort	27	1	2
	Square root	Square soot	80	1	2
	Binary search	Binary search	31	1	3
	Triangle classifier	Triangle classifier	70	2	4
2	OrdSet	Ordered set container	229	2	26
	Elevator	Elevator simulator	580	12	77
	Email-spl	Email tool	1233	17	331
	Minepump-spl	Mine pump simulator	580	15	203
	Replicatedworkers	Replicated workers simulator	342	14	47
3	Ant	Deployment tool	80,500	627	1762
	Jboss	Application server	116,638	1126	23,851
	Jmeter	Load test tool	43,400	389	2962
	Log4j3	Log tool for java	15,744	175	1752
	Xstream-spl	Serialise objects to XML	14,480	509	2382
	Jtopas	Text parsing	5400	50	568
	Nanoxml	XML parsing	7646	24	212
	XML-security	XML encryption	16,800	143	1321
	Siena	Event Notification	6035	26	113

of code (columns 3 and 4), number of classes (column 5) and number of methods (column 6). The sizes of the subjects in terms of the lines of code varied from 27 for Bubble sort to over 116,638 for Jboss.

For the first and the second groups, it was easier to analyses the source codes of the programs manually to find out the infeasible paths, and it was convenient to compare the results with the results obtained by our SMT-IPD. We used precision and recall rate to evaluate the experiment results.

To verify the effectiveness of our method, we designed a second experiment, where we chose the large-scale programs. We compared our method with the Java infeasible code detection tool Joogie [5].

To further verify the validity of our method, we designed a third experiment, where we chose four programs from the above 19 programs. The sizes of the programs in terms of the lines of code varied from 70 for the Triangle classifier to over 16,800 for XML-security. We compared our method with two other infeasible path detection approaches used in [6] and [7], respectively.

For the second and the third group of subjects, we chose the original version of OrdSet, Elevator, Email-spl, Minepump-spl and Replicatedworkers; the original version v_0 of Ant, Apollo, Jboss, Jmeter, Log4j3 and Xstream-spl; the original version v_3 of Jtopas; the original version v_5 of Nanoxml; the original version v_0 of Xml-security; and the original version v_4 of Siena for the experiments.

For the third group, we chose the Ant program, which is a Java-based build tool supplied by the open source Apache project. It has complex functions, and most of the infeasible paths detected came from the org.apache.tools.ant.taskdef package. Jboss and Jmeter are applications based on a network. Jboss is a web application server that contains many business-related judgment branches. The main function of Jmeter is stress testing for a web server, and there are many judgment branches of performance counters and time statistics. Log4j3 and Xstream-spl contain a large number of stream processing methods in Java. The former generates logs through the output stream, and the latter deals with the serialization and deserialization of objects. Jtopas, Nanoxml and XML-security are text parsing-related applications. JTopas is a Java library used for parsing text data. NanoXML is a small XML parser for Java. XML-security is a component library that implements XML signature and encryption standards; it is supplied by the XML subproject of the open source Apache project. Siena works as a network event notification system. This program is an Internet-scale event notification middleware for distributed event-based applications deployed over wide-area networks that is responsible for selecting notifications that are of interest to clients (as expressed in client subscriptions) and then delivering these notifications to the clients via access points.

C. EXPERIMENTAL RESULTS FOR THE FIRST AND THE SECOND GROUP OF PROGRAMS

Table 2 shows the results of the experiments for the small-scale programs. The table shows, for each subject, the group number, the program name (columns 1 and 2), the number of infeasible paths that SMT-IPD detected (column 3), the number of infeasible paths that we manually analyzed (column 4), the number of false-positive cases detected by SMT-IPD (column 5), and the detection precision and the recall rate of SMT-IPD (columns 6 and 7). Precision measures the percentage of the number of infeasible paths correctly detected by SMT-IPD. Recall measures the percentage between the number of infeasible paths correctly detected by our approach and the actual number of infeasible paths checked manually. Because precision can reflect the accuracy of the test results and recall rate can reflect the comprehensiveness of the test results, we use precision and recall as the indicators of results. The precision and recall rate are calculated according to

TABLE 2. Results for the first and the second group of programs.

Group	Program	$I_{detected}$	I_{true}	I_{false}	Precision,%	Recall,%
1	Zip viewer	2	5	0	100	40
	Bubble sort	2	2	0	100	100
	Square root	1	4	0	100	25
	Binary search	4	4	0	100	100
	Triangle classifier	41	41	0	100	100
2	OrdSet	38	38	0	100	100
	Elevator	22	31	0	100	71
	Email-spl	7	13	0	100	53.8
	Minepump-spl	4	4	0	100	100
	Replicatedworkers	34	34	0	100	100

TABLE 3. Experimental comparison results between SMT-IPD and Joogie for the third group of programs.

Program	SMT-IPD					Joogie					
	$M_{checked}$	$LOC_{checked}$	TP	$I_{detected}$	Time (s)	$M_{checked}$	$LOC_{checked}$	TP	$B_{detected}$	$I_{detected}$	Time (s)
Ant	85	1958	31,273	20,427	17,300	33	940	12,366	68	9437	4116
Jboss	885	19,782	1,440,524	1,333,837	7729	N	N	N	N	N	N
Jmeter	64	1269	11,076	8498	10,284	20	458	2741	33	1933	2632
Log4j3	63	742	1658	1247	3773	2	76	69	10	47	877
Xstream-spl	83	1229	43,245	41,797	3236	17	231	5792	27	4111	683
Jtopas	40	690	1054	749	1123	4	78	196	6	63	297
Nanoxml	21	541	8987	5393	1570	–	–	–	–	–	–
Xml-security	44	868	26,188	8914	3310	12	291	17,403	19	5610	834
Siena	3	33	46	21	1287	–	–	–	–	–	–

equations (1) and (2), respectively.

$$\Pr ecision = \frac{I_{\det ected} - I_{false}}{I_{\det ected}} \tag{1}$$

$$\text{Re } call = \frac{I_{\det ected} - I_{false}}{I_{true}} \tag{2}$$

where $I_{detected}$ denotes the number of infeasible paths detected by SMT-IPD, I_{false} denotes the number of infeasible paths that SMT-IPD is False Positives, and I_{true} denotes the number of infeasible paths that exist in the detected program.

As can be seen from the results in Table 2, the precision of the results generated by SMT-IPD was very high. For the programs Bubble sort, Binary search, Triangle classifier, OrdSet, Minepump-spl and Replicatedworkers, they contained pure numerical comparisons in the branch conditions, which can be solved accurately by a constraint solver; therefore, the precision and recall rates were relatively high. However, for the programs Zip viewer, Square root, Elevator and Email-spl, they had a lower recall rate, which means that SMT-IPD detected part of the infeasible paths. After analyzing, we found that Zip viewer contains a file iterator judgment, Square root contains a judgment if a Java object is NULL or not, Elevator uses a finite-state machine to simulate an elevator and Email-spl contains some network-related judgments. These four types of constraints cannot be solved by a constraint solver. During the infeasible path detection, these constraints are difficult to solve [1].

D. EXPERIMENTAL COMPARISON RESULTS BETWEEN SMT-IPD AND JOOGIE

To evaluate the effectiveness of our method, we selected some programs for our experiments. Table 3 shows the results of the comparison between our method (SMT-IPD) and Joogie [5]. The table shows, for each subject, the program name (column 1), the detection results by SMT-IPD (columns 2-6), and the detection results by Joogie (columns 7-12). Column $\mathbf{M_{checked}}$ (checked methods) (column 2 and column 7) is the number of detected methods that contained infeasible paths. Column $\mathbf{LOC_{checked}}$ (checked lines of code) (column 3 and column 8) is the total number of source code lines of $\mathbf{M_{checked}}$. Column **TP** (total paths) (column 4 and column 9) is the total number of paths of $\mathbf{M_{checked}}$. Column $\mathbf{I_{detected}}$ (infeasible paths detected) (column 5) in sub-table SMT-IPD is the number of infeasible paths that SMT-IPD detected. Column $\mathbf{B_{detected}}$ (basic blocks detected) (column 10) in sub-table Joogie is the number of infeasible basic blocks detected by Joogie. Column $\mathbf{I_{detected}}$ (infeasible paths detected) (column 11) in sub-table Joogie is the number of infeasible paths based on our statistics that were contained in the infeasible basic blocks. Column **Time** (column 6 and column 12) is the time needed by a subject to complete the detection separately by SMT-IPD and Joogie.

Besides Siena program, the eight programs contained a number of strings or pure numerical comparison branches, which belong to data types that can be solved by a constraint solver. Therefore, it is possible to detect an impressive number of infeasible paths by our SMT-IPD. For the Siena program, although it contains a large number of branches and loops, the proportion of the constraints that could be solved was not high owing to the limitations of the constraint solver. Therefore, SMT-IPD only detected three methods that contained 21 infeasible paths.

Joogie [5] detected infeasible basic blocks in the programs. Compared with SMT-IPD, the detection granularity of Joogie was smaller. Joogie failed to test the subjects Siena and Nanoxml, denoted with "-" in Table 3. The reason was that

TABLE 4. Information about the sub-path set for the third group of programs.

Program	TSP	$FSP_{detected}$	$ISP_{detected}$	$UNSP_{detected}$	DCR, %
Ant	4690	374	3064	1252	37.0
Jboss	138,284	22,746	79,682	35,856	31.9
Jmeter	1098	192	581	325	34.5
Log4j3	202	25	136	41	22.9
Xstream-spl	2631	717	1311	603	32.8
Jtopas	110	7	67	36	23.6
Nanoxml	855	66	552	237	25.9
Xml-security	2530	334	903	1293	34.8
Siena	6	2	1	3	19.0

Joogie first converts a Java code into a Boogie code, and there were a number of Java statements (such as arrays) and data types that could not be converted to Boogie codes; therefore, the test failed. A null pointer exception occurred when Joogie was detecting Jboss; therefore, the result was null, denoted with ''N'' in Table 3.

Joogie detects infeasible basic blocks, whereas SMT-IPD detects infeasible paths. We counted the number of infeasible paths $\mathbf{I_{detected}}$ that passed through the infeasible basic blocks $\mathbf{B_{detected}}$, and we compared it with the number of infeasible paths detected by SMT-IPD. What is more, we also compared the number of methods that contained infeasible paths detected by Joogie with that detected by SMT-IPD. We can see that our method is much better than Joogie. For example, our method can handle subjects such as Siena, Jboss and Nanoxml that cannot be detected by Joogie. Therefore, SMT-IPD is more effective.

For the data in Table 3, we randomly selected 500 infeasible paths from 1,420,883 infeasible paths (detected by SMT-IPD) but did not analyses all of them and analyzed the causes of the conflicts manually. The analysis showed that SMT-IPD is quite accurate.

The column **Time** (column 6 and column 12) includes the following parts. The first part is the time spent analyzing a Java class. Soot took about 8s to analyses a class. The second part is the time spent in the sub-path set generation. We used a recursive algorithm to generate a sub-path set, which took about 500 ms per thousand of lines of code. The constraint solver took about 250 ms to solve a system of inequalities that contained five linear inequalities. Therefore, we can assume that determining the feasibility of a sub-path took 250 ms. The sub-path set expansion algorithm has a very high time complexity, and the structure and size of the test procedures also influence the time consumption. Therefore, it is difficult to estimate the average time consumption. We adopted a multi-thread technology to accelerate the expansion of a sub-path set. To observe and search the test results conveniently, we wrote the test results into a Sqlite database. Owing to the write speed limitation of a Sqlite database, this stage is very time consuming, which consumed about half of the total time of the experiments.

The experimental results in Table 3 show that the infeasible path was more than 50% ($I_{detected}$/TP) for the programs that included infeasible paths. Suppose we generated test cases for a method that included an infeasible path, it would lead to at least 50% wastage of test resources because of the infeasible path in the method. If we knew the feasibility of the paths, we could only generate the test case for the feasible path. Therefore, detection of an infeasible path exactly can effectively save resources for testing.

In summary, our method ensures the comprehensiveness of the test results at a reasonable time cost. To observe the time difference between writing the results to plain text files and that to a Sqlite database, we tried to write the results to plain text files instead of to a Sqlite database for the Jboss system. We can see from Table 3 that the time cost of Jboss was much less than those of other subjects with a similar size. This improvement saves a considerable amount of time.

Table 4 shows the information about the sub-paths that were included in the methods that contained infeasible paths. The table shows, for each subject, the program name (column 1), the number of sub-paths in the methods that contained infeasible paths (TSP, column 2), the number of feasible sub-paths ($FSP_{detected}$, column 3), the number of infeasible sub-paths ($ISP_{detected}$, column 4), the number of sub-paths whose feasibility was unidentified ($UNSP_{detected}$, column 5) and the percentage of infeasible paths that were detected for the second time in all infeasible paths (DCR, column 6). From Table 4, we can see that only 29% (the average of column DCR) of the infeasible paths were detected in the second detection. For the feasible paths, although the second detection consumed a certain amount of time, the results that were generated by SMT-IPD were test cases, which completed the work of the test data generation to a certain extent. To sum up, our method can detect infeasible paths accurately and effectively, and can reduce the number of paths that need to be tested.

E. EXPERIMENTAL COMPARISON RESULTS BETWEEN SMT-IPD AND OTHER METHODS

For the groups used in the experiment, we separately chose one program from the first and the second groups, and chose two programs from the third group of experimental subjects in Table 1. We compared our method with two infeasible path detection approaches such as those of Suhendra *et al.* [7] and Gong and Yao [6]. The comparative experimental results for different scale programs are shown in Table 5.

Table 5 shows, for each subject, the program name (column 1), the total number of paths detected (column 2), the percentage of the infeasible paths that were detected

TABLE 5. Experimental comparison results for different scale programs.

Program	TP	Suhendra [7],%	Gong [6],%	Ours,%
Triangle classifier	63	39.96	48.63	65.08
Elevator	56	13.96	33.94	39.29
NanoXML	8987	34.89	38.63	60.01
XML-security	26,188	19.00	30.09	34.04

by the method of Suhendra *et al.* to the total number of paths (column 3), the percentage of the infeasible paths that were detected by the method of Gong *et al.* to the total number of paths (column 4) and the percentage of the infeasible paths that were detected by our method to the total number of paths (column 5). Where, the results are the percentages of the infeasible paths that were detected by the different methods to the total number of paths (TP). The method of Suhendra *et al.* [7] did not detect infeasible paths spanning across loop iterations. Thus, it considered the CFG to be a directed acyclic graph (DAG), representing the body of a loop. Furthermore, it only kept track of pairwise 'conflicts' between branches/assignments, which were either Assignment-Branch (A-B) conflicts or Branch-Branch (B-B) conflicts.

The method of Gong and Yao [6] used the maximum likelihood estimation to obtain the branch correlations and then detected the infeasible paths. However, there are some differences between our method and that of Gong and Yao [6]: (1) the method of Gong and Yao [6] does not analyses the A-B correlation information, and it can only detect part of the infeasible paths that were caused by all of the A-B conflicts. (2) When detecting the feasibility of paths for the open-source projects, the method of Gong and Yao [6] limited the maximum of the branch statements to those that had a B-B correlation and it could not detect the feasibility of paths that were caused by the branch statements that exceeded the limitation. In contrast, our method does not set a limitation, that is, it can detect the infeasible paths that were caused by any branch statements that have a B-B correlation and, thus, it has stronger applicability.

V. RELATED WORKS

There are many research studies in the area of detecting infeasible paths that are related to our work. Gong and Yao [6] and Suhendra *et al.* [7] proposed several methods based on branch correlation. These methods cannot detect complex predicates accurately. Unlike their approach, our method uses an SMT solver to solve predicates, and the test results are more accurate for first-order predicates.

Delahaye *et al.* [8], Jaffar *et al.* [9] and Tomb and Flanagan [10] detected infeasible paths based on the satisfiability of the path conditions. These studies used methods similar to a symbolic execution to extract path conditions and determine whether a path was infeasible or not. However, such a method needs to analyses all target paths, which consumes considerable test resources. Junker *et al.* [11] approach was to view static analysis as a model checking problem, to iteratively compute infeasible sub-paths of infeasible paths using SMT solvers. Our approach mainly detects the sub-path set and reduces the cost of analysis.

Hermadi *et al.* [12], Ghiduk [13] and Tonella *et al.* [14] proposed dynamic methods for infeasible path detection, which try to generate test cases for paths using a limited-depth search. If test cases could not be found to cover a specific path, this path was marked as infeasible. In path testing, such methods are very effective. However, if all the paths in a program need detect, the cost will be extremely high owing to the path-explosion problem. Unlike their approaches of the references [12]–[14], the detection range of our method is more comprehensive. Most of the detection works are on the sub-path set. Our method can solve the path-explosion problem effectively and determine the feasibility of all paths.

Ngo and Tan [15] proposed a hybrid method that combined a static method based on branch correlation analysis and a dynamic method that chose feasible paths positively by searching for appropriate test cases. This method achieved satisfactory detection results, but consumed considerable test resources because the detection combined static analysis and dynamic analysis. This method is not suitable for detecting large programs. However, our method is a static analysis method that does not execute any programs, which can avoid the overhead work required for dynamic analysis.

Lipton *et al.* [16] and Hamlet [17] were the first to propose equivalent mutants. They obtained a mutated program by using syntax mutation operators to modify the program slightly. Then, they ran test cases on the mutated program and the original program to compare the similarities and differences between the results. Gong *et al.* [18] pointed out that for a mutated statement, if there was no variable reference in the consequent code, or the effect could not be propagated, its mutant branch was infeasible. Offutt and Pan [19], [20] addressed the problem of infeasible path detection by detecting equivalent mutants; however, the cost of this method was expensive because plenty of mutated programs should be run. In contrast, our method is static and, therefore, the cost is relatively small. In addition, DeMillo and Offutt [21] pointed out that whether the mutated program was equivalent to the source program could not be determined in the mutation testing. Therefore, an infeasible path detection method based on equivalent mutants is relatively inaccurate.

Zhang *et al.* [22] presented a novel method to generate test data covering many paths of a complicated program. Their generation of test data covering many paths is formulated as several sub-optimization problems by grouping. Gong *et al.* [23] focused on the problem of reducing scheduling sequences for statement coverage of message-passing parallel programs. Tian and Gong [24] focused on generating test data covering paths rather than investigating coverage criteria. Instead of seeking for paths, they set a path as the target in advance. They studied effective methods of generating test data by using the co-evolutionary genetic algorithm and the features of parallel programs. However, our method

is a static analysis method that does not need to execute programs, which can avoid the overhead work required for execute programs.

VI. THREATS TO VALIDITY

We have noticed that there are two threats to the validity of our approach.

The first threat is the accuracy of our approach, which is affected mainly by two factors: the accuracy of the path condition analysis and the accuracy of the SMT solver.

Aiming at the path condition analysis factor, we extract path conditions by analyzing the code written in the Jimple intermediate language, which is transformed from Java bytecodes by Soot. The Jimple code is a static single-assignment form; therefore, the complicated path conditions can be split into several static single-assignment sub-conditions. Our approach analyses each sub-condition and summarizes them to obtain the complete path conditions; therefore, the process of path condition analysis is accurate.

As for the accuracy of the SMT solver factor, the SMT solver has higher accuracy[6] for first-order equations. We manually verified 500 infeasible paths and found no errors. Some SMT solvers provide support for multi-order equations, such as Choco.[7] Choco solves a problem by alternating the constraint filtering algorithms with a search mechanism. We chose SMT Interpol for our experimentation because it is one of the fastest SMT solvers written in Java.[8] However, owing to the limitations of the SMT solver, we can only deal with numerical comparison branches. Most programming languages are object oriented (like Java); therefore, most of the function calls contain object references, which are not supported by an SMT solver. In this case, we treat the feasibility as unidentified. Therefore, our method cannot detect inter-procedural infeasible paths.

The second threat is the path-explosion problem. In Section III.D, we can see that a certain infeasible sub-path (like sp_4) can be expanded into several infeasible paths. In a program, a conflict point may cause many infeasible paths. If a certain infeasible sub-path contains a conflict point, then this sub-path can be expanded into a number of infeasible paths. To address this problem, we adopted the sub-path expansion mechanism, which can solve the path-explosion problem partly.

VII. CONCLUSIONS

This paper proposes a static method (SMT-IPD) based on an SMT solver for detecting infeasible paths. This method first generates a sub-path set, analyses branch predicates and correlation variable *definition* points to construct a system of inequalities, and then uses a constraint solver to solve the inequalities. According to the results solved by the SMT solver, SMT-IPD determines whether a sub-path is infeasible. If the sub-paths are infeasible, then the expanded paths from them are infeasible too. The paths that were expanded from the undetermined part of the sub-paths need to be tested again to determine their feasibility. Finally, the results contain the feasibility of all the paths. Our experimental results showed that this method has a higher accuracy, which can detect infeasible paths effectively.

This study, like any other empirical study, has some limitations. The first limitation is that our method cannot detect inter-procedural infeasible paths. The second limitation is that we don't consider polymorphism, which may influent the accuracy. The third limitation is the way of dealing with the cycles, which are some assumptions we made in Section III. A. This method indeed may affect the accuracy of the approach. That is because our method is static analysis, and static analysis methods generally deal with loops like this. It's really not as accurate as that of obtained by actually running the program. In future work, we plan to apply some intelligent judgment strategy to the sub-path set construction algorithm, to obtain as many infeasible paths as possible. We also plan to extend the optimal algorithm for generating the sub path set, etc. Moreover, we will plan to consider polymorphism in order to improve the accuracy.

[6]http://smtcomp.sourceforge.net/2015/results-QF_LRA.shtml?v=1446209369

[7]http://choco-solver.org/

[8]http://smtcomp.sourceforge.net/2015/results-summary.shtml?v=1446209369

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers and editors for suggesting improvements and for their very helpful comments.

REFERENCES

[1] S. Ding and H. B. K. Tan, "Detection of infeasible paths: Approaches and challenges," in *Proc. Int. Conf. Eval. Novel Approaches Softw. Eng.*, 2013, pp. 64–78.

[2] K. Kempf and F. Slomka, "Direct handling of infeasible paths in the event dependency analysis," in *Proc. 20th IEEE Int. Conf. Embedded Real-Time Comput. Syst. Appl.*, 2014, pp. 1–10.

[3] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 125–132, Jul. 1987.

[4] H. Agrawal, "Dominators, super blocks, and program coverage," in *Proc. 21st ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang.*, Feb. 1994, pp. 25–34.

[5] S. Arlt and M. Schäf, "Joogie: Infeasible code detection for java," in *Proc. Int. Conf. Comput. Aided Verification*, 2012, pp. 767–773.

[6] D. Gong and X. Yao, "Automatic detection of infeasible paths in software testing," *IET Softw.*, vol. 4, no. 5, pp. 361–370, Oct. 2010.

[7] V. Suhendra, T. Mitra, and A. Roychoudhury, "Efficient detection and exploitation of infeasible paths for software timing analysis," in *Proc. 43rd ACM/IEEE Design Automat. Conf.*, Jul. 2006, pp. 358–363.

[8] M. Delahaye, B. Botella, and A. Gotlieb, "Explanation-based generalization of infeasible path," in *Proc. 3rd Int. Conf. Softw. Test., Verification Validation*, Apr. 2010, pp. 215–224.

[9] J. Jaffar, V. Murali, J. A. Navas, and A. E. Santosa, "TRACER: A symbolic execution tool for verification," in *Proc. Int. Conf. Comput. Aided Verification*, 2012, pp. 758–766.

[10] A. Tomb and C. Flanagan, "Detecting inconsistencies via universal reachability analysis," in *Proc. Int. Conf. Softw. Test. Anal.*, Jul. 2012, pp. 287–297.

[11] M. Junker, R. Huuck, A. Fehnker, and A. Knapp, *SMT-Based False Positive Elimination in Static Program Analysis* (Lecture Notes in Computer Science book series). vol. 7635. Berlin, Germany: Springer-Verlag, 2012.

[12] I. L. C. Hermadi and R. Sarker, "Dynamic stopping criteria for search-based test data generation for path testing," *Inf. Softw. Technol.*, vol. 56, no. 4, pp. 395–407, Apr. 2014.
[13] A. S. Ghiduk, "Automatic generation of basis test paths using variable length genetic algorithm," *Inf. Process. Lett.*, vol. 114, no. 6, pp. 304–316, Jun. 2014.
[14] P. Tonella, R. Tiella, and C. D. Nguyen, "Interpolated n-grams for model based testing," in *Proc. Int. Conf. Softw. Eng.*, May 2014, pp. 562–572.
[15] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," *Inf. Softw. Technol.*, vol. 50, nos. 7–8, pp. 641–655, Jun. 2008.
[16] R. A. DeMillo, R. J. Lipton, and F. G. Sayward, "Hints on test data selection: Help for the practicing programmer," *Computer*, vol. 11, no. 4, pp. 34–41, Apr. 1978.
[17] R. G. Hamlet, "Testing programs with the aid of a compiler," *IEEE Trans. Softw. Eng.*, vol. SE-3, no. 4, pp. 279–290, Jul. 1977.
[18] D. Gong, G. Zhang, X. Yao, and F. Meng, "Mutant reduction based on dominance relation for weak mutation testing," *Inf. Softw. Technol.*, vol. 81, pp. 82–96, Jan. 2017.
[19] A. J. Offutt and J. Pan, "Automatically detecting equivalent mutants and infeasible paths," *Softw. Test., Verification Rel.*, vol. 7, no. 3, pp. 165–192, Sep. 1997.
[20] A. J. Offutt and J. Pan, "Detecting equivalent mutants and the feasible path problem," in *Proc. 11th Annu. Conf. Comput. Assurance (COMPASS)*, Jun. 1996, pp. 224–236.
[21] R. A. DeMilli and A. J. Offutt, "Constraint-based automatic test data generation," *IEEE Trans. Softw. Eng.*, vol. 17, no. 9, pp. 900–910, Sep. 1991.
[22] W. Q. Zhang, D. W. Gong, and X. J. Yao, "Evolutionary generation of test data for many paths coverage based on grouping," *J. Syst. Softw.*, vol. 84, no. 12, pp. 2222–2233, Dec. 2011.
[23] D. Gong, C. Zhang, T. Tian, and Z. Li, "Reducing scheduling sequences of message-passing parallel programs," *Inf. Softw. Technol.*, vol. 80, pp. 217–230, Dec. 2016.
[24] T. Tian and D. Gong, "Test data generation for path coverage of message-passing parallel programs based on co-evolutionary genetic algorithms," *Automated Softw. Eng.*, vol. 23, no. 3, pp. 469–500, Sep. 2016.

SHUJUAN JIANG was born in Laiyang, Shandong, in 1966. She received the B.S. degree from the Computer Science Department, East China Normal University, Shanghai, in 1990, the M.S. degree from the China University of Mining and Technology, Xuzhou, Jiangsu, in 2000, and the Ph.D. degree from Southeast University, Nanjing, Jiangsu, in 2006, all in computer science.

Since 1995, she has been a Teaching Assistant, a Lecturer, an Associate Professor and a Professor with the Computer Science Department, China University of Mining and Technology. She has authored ten books, more than 80 articles, and more than six inventions. Her research interests include software engineering, program analysis and testing, and software maintenance.

HONGYANG WANG was born in Yichun, Heilongjiang, China, in 1990. He received the B.S. degree in computer science from the Computer Science Department and the M.S. degree in computer science from the China University of Mining and Technology, Xuzhou, Jiangsu, in 2013 and 2016, respectively.

Since 2016, he has an Employee at Payment Technology Company Ltd. He has authored over two articles. His research interests include software engineering, and software analysis and testing.

YANMEI ZHANG was born in Tangshan, Hebei, in 1982. She received the B.S. degree in computer science from the Computer Science Department, Nanjing Tech University, in 2007, and the M.S. and Ph.D. degrees in computer science from the China University of Mining and Technology, Xuzhou, Jiangsu, in 2009 and 2012, respectively. From 2012 to 2017, she was a Lecturer, and since 2018, she has been an Associate Professor with the School of Computer Science, China University of Mining and Technology. She has authored one book, more than ten articles, and more than two inventions. Her research interests include software engineering, program analysis and testing, software quality, and intelligent evolutionary algorithm.

MENG XUE was born in Pizhou, Jiangsu, in 1979. He received the B.S. and M.S. degrees in computer science from the Computer Science Department, China University of Mining and Technology, Xuzhou, Jiangsu, in 2001 and 2004, respectively.

Since 2001, he has been a Teaching Assistant and a Lecturer with the Computer Science Department, China University of Mining and Technology. He has authored two books and more than ten articles. His research interests include program testing, software maintenance, and image compression.

JUNYAN QIAN was born in Shaoxing, Zhejiang, in 1973. He received the B.S. degree from Anhui Polytechnic University, China, in 1996, the M.S. degree from the Guilin University of Electronic Technology, China, in 2000, and the Ph.D. degree in computer science from Southeast University, Nanjing, Jiangsu, in 2008.

He has been a Professor with the School of Computer Science and Engineering, Guilin University of Electronic Technology, China. His research interests include formal verification, optimization algorithm, and reconfigurable VLSI design.

MIAO ZHANG was born in Zoucheng, Shandong, in 1992. She received the B.S. degree in computer science from the Computer Science Department, Shandong University, Jinan, Shandong, in 2014. She is currently pursuing the M.S degree with the School of Computer Science and Technology, China University of Mining and Technology, Xuzhou, Jiangsu.

She has authored more than two articles. Her research interests include software analysis and testing, and software maintenance.