Detecting Infeasible Branches Based on Code Patterns

Sun Ding
Nanyang Technological University
Singapore 639798
ding0037@ntu.edu.sg

Hongyu Zhang
Tsinghua University
Beijing 100084, China
hongyu@tsinghua.edu.cn

Hee Beng Kuan Tan
Nanyang Technological University
Singapore 639798
ibktan@ntu.edu.sg

***Abstract*—Infeasible branches are program branches that can never be exercised regardless of the inputs of the program. Detecting infeasible branches is important to many software engineering tasks such as test case generation and test coverage measurement. Applying full-scale symbolic evaluation to infeasible branch detection could be very costly, especially for a large software system. In this work, we propose a code pattern based method for detecting infeasible branches. We first introduce two general patterns that can characterize the source code containing infeasible branches. We then develop a tool, called Pattern-based method for Infeasible branch Detection (PIND), to detect infeasible branches based on the discovered code patterns. PIND only performs symbolic evaluation for the branches that exhibit the identified code patterns, therefore significantly reduce the number of symbolic evaluations required. We evaluate PIND from two aspects: accuracy and efficiency. The experimental results show that PIND can effectively and efficiently detect infeasible branches in real-world Java and Android programs. We also explore the application of PIND in measuring test case coverage.**

***Index Terms*—Infeasible branch, test case coverage, code patterns, program analysis.**

I. Introduction

In order to test a program more effectively, it is desirable to have matured tools that can automatically generate test cases and achieve high test coverage. One major barrier is the existence of infeasible branches, which are program branches that can never be exercised regardless of the inputs of the program. Figure 1 shows a real Java program of the Apache Struts framework. It contains an example of infeasible branch. The reaching definition of *ttype* at line 2 is a constant. Therefore, the *if* predicate at line 2 is always evaluated to *false*, causing the true branch of the predicate (from line 2 and line 3) to be infeasible.

Detecting infeasible branches is important to test case coverage calculation [28] and test case generation [2, 3]. For instance, in Figure 1, if a developer submits a test case (*begin*=0, *textLength*=10) to EclEmma [1] (a popular code coverage tool widely used for testing Java programs), EclEmma will report that this test case fails to cover one branch and the branch coverage is only 50%. This result may be misleading as more test cases could be attempted to enhance the coverage. However the missed branch is actually infeasible and cannot be exercised by any input. Therefore, resources spent on improving test coverage could be wasted.

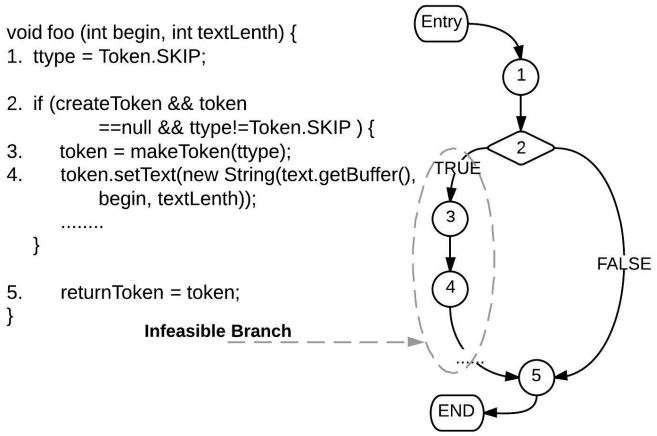

Fig. 1. An example of an infeasible branch

The existence of infeasible branches also adversely affects many other software engineering tasks such as program understanding and maintenance. The code about infeasible branch is often confusing and misleading, which could cause maintenance problems as the real intention of programmers who wrote this code is obscure. Therefore, it is useful to detect infeasible branch occurrences in programs in order to improve code quality and maintainability.

To detect infeasible branches, one solution is to perform full-scale symbolic evaluations for all branches in a program and to evaluate their satisfiability using a constraint solver such as Z3 [4]. The drawback of this method is that it has to evaluate all program branches, which is computationally expensive, especially for a large-scale system.

Through empirical analysis, we discover two code patterns that can generalize the occurrence of infeasible branches, namely (1) Constant Value pattern and (2) Semantic Dominator pattern. These patterns are heuristic patterns that are obtained through empirically studying source code containing infeasible branches. By searching for these two code patterns, we can detect infeasible branches efficiently. In our method, we only evaluate the branches that exhibit the identified patterns, therefore, significantly reducing the number of symbolic evaluations required.

We develop a tool, called Pattern-based method for Infeasible branch Detection (PIND), to detect infeasible branches based on the discovered patterns. We have evaluated

PIND on a number of Java and Android systems and obtained promising results in terms of accuracy and efficiency. For detecting infeasible branches in the evaluated systems, PIND achieves an average precision of 100%, and an average recall of 94.92% in an intra-procedural analysis. PIND also achieves an average precision of 100% and an average recall of 98.15% in an inter-procedural analysis. Furthermore, PIND only requires 34% and 40.72% of the processing time that an exhaustive symbolic evaluation costs in the intra- or inter-procedural analysis, respectively.

The contributions of this work are as follows:

- We propose a novel, code pattern based technique for detecting infeasible branches. To our best knowledge, this is the first time such a technique is proposed.
- We have evaluated the correctness and efficiency of the proposed tool on real-world Java and Android programs. The results have confirmed the effectiveness of the proposed tool.
- We point out the application of the proposed method in computing test case coverage.

The remainder of this paper is organized as follows: Section II introduces two patterns of infeasible branches. Section III proposes a pattern-based approach for detecting infeasible branches. Section IV presents our experimental design and shows the experimental results. We discuss the application of the proposed approach in Section V and threats to validity in Section VI. Section VII surveys related work followed by Section VIII that concludes this paper.

II. Empirical Patterns of Infeasible Branch

A. Basic Terms

In this paper, we follow the formalism of a **control flow graph (CFG)** [5] and the related classic concepts and terms for control and data dependency. Additionally, we define a transitive relationship.

A node y in a CFG **transitively references** to a variable V defined at node x if there is a sequence of nodes, $x = y_0, y_1, \ldots, y_n = y$, and a sequence of variables $V = V_0, V_1, \ldots, V_n$, such that $n \geq 1$, for each j, $1 \leq j \leq n$, y_j defines variable V_j and references to variable V_{j-1}, and any sub-path from y_{j-1} to y_j without passing through y_{j-1} again is a definition-clear path with respect to $\{V_{j-1}\}$. Furthermore, we say that node y is transitively control dependent on node x if there is a sequence of nodes, $x = x_0, x_1, \cdots, x_n = y$, such that x_j is **control dependent on** x_{j-1}, $1 \leq j \leq n$.

An **external input** refers to one of the following sources:

1) Parameter
2) Return value of a function call
3) Variable which is modifiable by a function call

In a CFG, an arbitrary variable x could only be either of the following types:

1) x is data dependent on a set of external inputs. We call this type as external-sensitive variable.

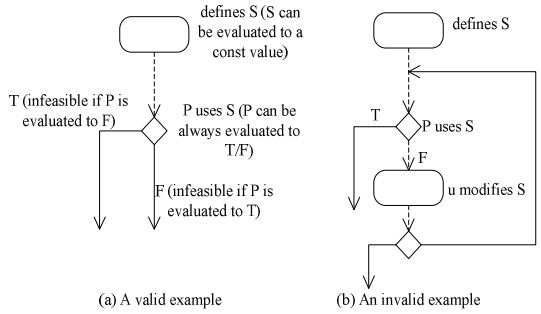

Fig. 2. The Constant Value pattern

```
1  boolean dothis = false;
2  if (dothis)  {
3    if (closeDecl) {
4      writer.write('>');
5      writer.write(m_lineSep, 0, m_lineSepLen);
   }
  }
```

Fig. 3. A code example of Constant Value pattern

2) x is not data dependent on any external input. We call this type as **external-free variable**.

We call a node in a CFG that only references to external-free variables an **external-insensitive node**.

B. Constant-Value Pattern

In this sub-section, we introduce the first pattern—Constant-Value Pattern.

Description: Let b be a branch at predicate node P in a CFG, let D be b's branch condition at P, if the following statements hold, b is an infeasible branch, if and only if:

1) P is an external-insensitive node.
2) P is not data dependent on any of its successor.
3) D can be always evaluated as unsatisfiable.

The Constant-Value pattern is illustrated in Figure 2 (a). The example in Figure 2 (b) is invalid because the clause (3) is not satisfied – P resides in an iteration structure and there is a path from u to P (u changes the value of S) so that P is data dependent on u. Therefore, P may be evaluated to different values in different iterations.

Fig. 3 shows a code fragment. The predicate at line 2 uses the variable *dothis*, whose value can be always evaluated to *false*. Therefore, the predicate is always evaluated to *false*, which means the *true* branch from line 2 to line 3 is an infeasible branch. Similarly, the example in Figure 1 is also an instance of the Constant-Value pattern.

C. Semantic Dominator Pattern

In this sub-section, we introduce the second pattern—Semantic Dominator Pattern.

Description: Let b be a branch at predicate node P in a CFG, let D be b's branch condition at P, if the following statements hold, b is an infeasible branch, if and only if:

1) There are a non-empty set of predicates $S = \{Q_i \mid Q_i$ dominates P AND P is transitively control dependent on $Q_i\}$

2) For any two predicates within $S \cap \{P\}$, they transitively reference to a common set of external sensitive variables.

3) If there is one back edge from P to any $Q_i \in S$, then excluding P and Q_i, P is not data dependent on any nodes on this back edge.

4) There is a path μ that follows P's branch b and contains all the predicates in S .

5) Along μ, by joining the branch condition of each Q_i and b, the conjunction can always be evaluated as *unsatisfiable*.

The Semantic Dominator pattern is illustrated in Figure 4 (a). The example in Figure 4 (b) is invalid because P and Q are sequential statements and the basis paths from P to Q could follow two branches of Q. Therefore the semantic of Q does not dominate P.

Fig. 5 gives an example of the Semantic Dominator pattern. The predicates at line 2 and line 4 all reference to the same set of variable: {*cacheLen*}. The *false* branch's condition at line 4 can be always evaluated to *unsatisfiable* based on the predicate at line 4. Therefore, the *false* branch of line 4 is an infeasible branch.

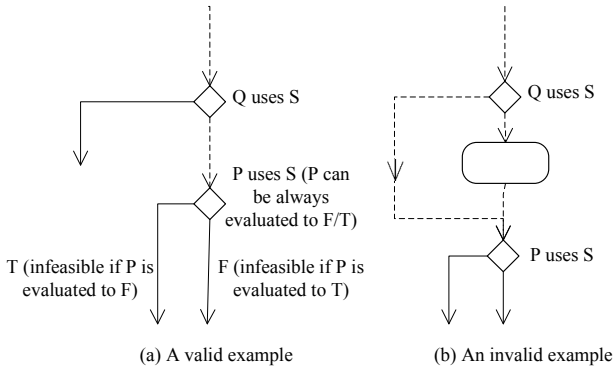

Fig. 4. The Semantic Dominator pattern

```
1  ......
2  if ((cacheLen > 0) && (Y != NULL) {
3    count += (cacheLen + counter.m_countNodesStartCount);
4    if (cacheLen > 0)
5      appendBtoFList(counter.m_countNodes,m_newFound);
6    m_newFound.removeAllElements();
7    return count;
   }
8  ......
```

Fig. 5. A code example of Semantic Dominator pattern

III. Pattern-based Method for Infeasible Branch Detection

A. Overview

We develop a tool, called ***P**attern-based method for **In**feasible branch **D**etection* (**PIND**), to recognize the two proposed patterns and detect infeasible branches. Figure 6 shows the overall structure of PIND. The major steps are summarized as follows:

PIND firstly applies syntax analysis to scan the occurrence of candidate predicates. For the Constant Value pattern, it searches for a single predicate that satisfy clauses (1) - (2) described in Section II.B. For the Semantic Dominator pattern, it searches for a pair of predicates that satisfy clauses (1) - (5) described in Section II.C.

Extracting the constraints to be evaluated from the candidate predicates. For the Constant Value pattern, the constraint is introduced by clause (3) described in Section II.B. For Semantic Dominator pattern, the constraint is introduced by clause (5) described in Section C.

After extracting the constraint, PIND applies symbolic evaluation to validate the constraint. If the validation result is *unsatisfiable*, the infeasibility of the corresponding branch can then be determined.

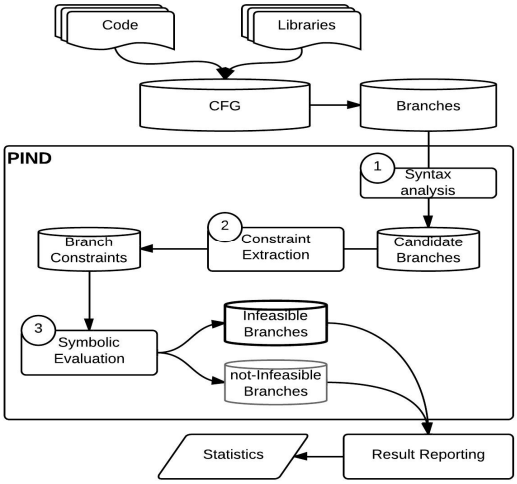

Fig. 6. The overall structure of PIND

Compared with the conventional, full-scale symbolic evaluation method, our method only performs symbolic evaluation for predicates that satisfy the discovered patterns. Therefore, we only evaluate a small percentage of total branches. Predicates that do not satisfy the properties of the patterns are not evaluated. In this way, our method saves expensive computation resources for symbolic evaluation and is more efficient.

B. Demand-driven Symbolic Evaluation

In order to achieve high accuracy, the symbolic evaluation used in PIND is path sensitive. However, an arbitrary path sensitive analysis is challenging as there could exist a huge number paths in a program. Therefore, PIND builds a path sensitive symbolic evaluation under a demand-driven framework [6, 7].

Demand-driven framework is an efficient framework of computing path-related properties that satisfying certain constraints. The idea is to model a query for a particular property at a statement of interest. The query is propagated backward along the path until it is resolved, or the propagation reaches the path head. Demand-driven based analysis only process limited paths and path nodes. So it is efficient and scalable to encounter large programs. It is proved that the worst case time complexity of demand-driven analysis is the same as that of the traditional iterative

framework, and in most cases, demand-driven based analysis is more efficient [7].

In order to evaluate a constraint, PIND encapsulates the constraint into a 3-tuple query $<\theta, \alpha, \beta>$, where:

1) θ is the constraint for symbolic evaluation. More specifically:

- For the Constant-Value pattern, θ is the constraint introduced by the pattern's clause(3).
- For the Semantic Dominator pattern, θ is the constraint introduced by the pattern's clause(5).

2) α is the predicate node where the query is raised.

3) β is the current node where the query is propagated to.

After raising a query at α, PIND constructs a set of paths. Each of such path, called candidate path, starts at the CFG entry node and ends at α, and passes through all the nodes that α is data dependent on. For each candidate path, PIND executes the following tasks:

1) PIND propagates the query backward along the path.

2) For each traversed node β, if the node contains a definition for any variable referenced by the query's constraint, PIND updates the query by adding the definition into θ.

3) PIND tries to validate the query once θ is updated. The validation is by submitting θ to an SMT solver Z3 [4].

For each candidate path, PIND recursively repeats the above steps until Z3 concludes θ as unsatisfiable on this path, or the propagation reaches the entry of the path. The query is concluded unsatisfiable when θ is unsatisfiable on all the candidate paths. Otherwise, the query is deemed satisfiable.

Some detailed designs of our demand-driven framework are as follows:

Symbolic evaluation: PIND encodes a query in the format of JIMPLE, an intermediate representation upon Soot's infrastructure [8]. Once a query's constraint θ is ready for submission to Z3, PIND encodes the constraint in the format of SMT-LIB2 [4], which supports primary types, array, fixed-point bit-vectors, and uninterpreted functions. PIND does not equip any ad-hoc engine of symbolic evaluation. Instead, it uses Z3 to handle constraint solving during the symbolic evaluation [2]. Further, PIND encodes constraints that contain only integer variables and integer constants with fixed-point bit-vectors to prevent possible inaccuracy caused by integer overflow [9].

Alias analysis: PIND follows a commonly-used way to handle alias information [7, 10]. Before PIND performs any demand-driven symbolic evaluation, the alias information is calculated by Soot's component Spark [10] in advance. PIND later uses the calculated alias information during the query propagation [7].

Function calls: To deal with function calls, PIND maintains a lookup table to store post-conditions of a particular function. This table has two fields: *function AST*, which records the syntax of a function; and *post-Condition,* which records the corresponding post-condition after executing this function. PIND supports both intra- and inter-procedural analysis as follows:

1) In an intra-procedural analysis, when a query is propagated to a function invoking the node, PIND joins the available post-conditions to the current query constraint. Otherwise, PIND treats the function as an uninterpreted function [4].

2) In an inter-procedural analysis, PIND uses the same way to deal with functions from native libraries and interface functions.

Test code: //test the true branch of node 2

```
1.  boolean dothis = false;
2.  if (dothis)  { // Candidate predicate P
3.    if (closeDecl) {
4.     writer.write('>');
5.     writer.write(m_lineSep, 0, m_lineSepLen);
        }
      }
```

Pattern Recognition:

- Perform syntax analysis and know that the test code satisfy clauses *(1)* to *(2)* of the Constant-Value pattern.
- Extract constraint as: *θ:=(dothis ==true)*
- Rise a query at node 2: *<θ, node2, node2>*
- Construct a set of candidate path: {*path1(node1, node2)*}
- Validate the query with symbolic evaluation: {*dothis =false && dothis==true*} → Z3
- **Z3** validation result: *Not valid*

Final Conclusion: the code contains Constant Value Pattern and the true branch of the predicate at line 2 is infeasible.

Fig. 7. An example of recognizing Constant-Value pattern

Test code:

//test the false branch of node 4

```
1. void functionX (int lenth, Object Y, Clock counter){
2.     if ( lenth > 0 ) {                    // Candidate predicate Q
3.         Y.count += (lenth + counter.m_countNodesStartCount);
4.         if (lenth > 0 &&  Y != NULL)   // Candidate predicate P
5.             appendBtoFList(counter.m_countNodes,m_newFound);
6.         m_newFound.removeAllElements();
7.         return count;
       }}
```

Pattern Recognition:

- Perform syntax analysis and know that the test code satisfy clauses *(1)* to *(4)* of the Semantic Dominator Pattern.
- Extract constraint as: *θ:= (lenth > 0) && ((lenth >0 &&* Y != NULL*)==false))*
- Rise a query at node 4: *<θ, node2, node4>*
- Construct a set of candidate path: {*path1(node1, node2,node3,node4)*}
- Validate the query with symbolic evaluation: *(lenth > 0) && ((lenth >0 && Y != NULL)==false)&& Y.count += (lenth + counter.m_countNodesStartCount))* → Z3
- **Z3** validation result: *Not valid*

Final Conclusion: the code contains Semantic Dominator Pattern and the false branch of the predicate at line 4 is infeasible.

Fig. 8. An example of recognizing Semantic Dominator Pattern

C. Examples

Fig. 7 shows an example of recognizing Constant-Value pattern. PIND first performs syntax analysis and detects the node at line 2 as a candidate of Constant Value pattern in this code. An query is raised as *<θ:=(dothis ==true), node2,*

node2> at node2. PIND constructs a set of candidate path, which is {*path1(node1, node2)*}. The query is propagated backward along the candidate path. After symbolic evaluation, the query is evaluated as unsatisfiable. Therefore, PIND confirms the true branch at line 2 as an infeasible branch.

Fig. 8 shows an example of recognizing Semantic Dominator pattern. PIND first performs syntax analysis and detects a pair of predicates {node2, node4} as a candidate of Semantic Dominator pattern. A query is raised at node 4 as <*θ:= (lenth > 0) && ((lenth >0 &&* Y != NULL*)==false)), node2, node4*>. PIND constructs a set of candidate path, which is {*path1(node1, node2,node3,node4)*}. The query is propagated backward along the candidate path. After symbolic evaluation, the query is evaluated as unsatisfiable. Therefore, PIND confirms that the false branch of the predicate at line 4 is infeasible.

IV. EVALUATION

A. Experiment Design

We conduct experiments to evaluate PIND. We select 3 standard open source Java programs and 3 Android programs [11] as subjects. Table I shows the statistics of the subject systems. The columns #*Total Method*, *KLOC* and *Cyclomatic* shows the number of total methods, thousands of lines of code (KLOC), and cyclomatic complexity metric [12] of each selected system, respectively. Furthermore, the column #*Root Method* shows the number of methods without any callers.

Because it is generally difficult to find out all false-negatives, we read the source code, manually sample a set of infeasible branches in each subject system and treat them as seeds. The number of sampled infeasible branches for each system is shown in Table IV and Table V. These sets of infeasible branches are considered as "ground truths" for evaluation purpose.

PIND supports both intra- and inter-procedural analysis. We conduct experiments on intra-CFG and inter-CFG separately. For the former, we verify PIND against the intra-CFGs of all the method in the selected systems. For the latter, we construct inter-CFGs from all the root methods in the selected systems and we only verify branches that contain at least one function call.

TABLE I. STATISTICS OF THE SUBJECT SYSTEMS

Systems	#Total Method	#Root Method	KLOC	Cyclomatic
Java				
JLibSys	669	392	10.65	2,150
PMD	5,032	1509	58.54	10,459
JMathLib	3,268	1029	53.48	2,993
Android				
UnifyUnitConverter	3,839	1139	N.A[1]	5,478
GoLocker	6,923	2071		10,441
GPSLogger	4,058	1405		8,741

[1] We analyze directly the Android APK files therefore we do not count the KLOC of their source code.

To better evaluate the effectiveness of PIND, we compare it with the conventional infeasible branch detection method that does full-scale symbolic evaluation for all branches without applying any patterns. We call the conventional method *FullZ3*. We implement this method by instrumenting Z3 for full-fledged symbolic evaluation. FullZ3 identifies a branch as (i) infeasible, (ii) feasible, or (iii) unknown.

Our objectives are to evaluate the *accuracy* and *efficiency* of PIND.

Accuracy evaluation: The accuracy is evaluated as follows:

1) We evaluate each selected system against PIND and FullZ3 independently and record the detected infeasible branches reported by them separately.

2) If a branch's infeasibility is deemed differently by the PIND and FullZ3, we will manually confirm the infeasibility. If an infeasible branch is wrongly detected as not-infeasible (feasible or unknown) by an approach, a false-negative is counted for that approach; if a non-infeasible branch is miss-detected as infeasible by an approach, a false-positive is counted for that approach.

3) Additionally, if any seeded infeasible branch is missed by an approach, a false-negative is counted for that approach.

Efficiency evaluation: The time cost is compared between PIND and the full-scale evaluation approach FullZ3. We also measure the computational cost in terms of the number of predicates been processed by PIND and FullZ3. The experiments are carried out on a PC with an Intel Core i5-M430-2.27GHz CPU and 4 GB memory.

TABLE II. RESULTS OF AN INFEASIBLE BRANCH DETECTOR

		Actual	
		Infeasible	Not- Infeasible
Detected	Infeasible	True positive (t_p)	False positive (f_p)
	Not-infeasible	False negative(f_n)	True negative (t_n)

B. Evaluation Metrics

Accuracy: The results of an infeasible branch detector can be summarized in Table II. To measure its Accuracy, we use the standard Recall and Precision metrics, which is shown by Equation (1). Precision (p_r) measures the actual infeasible branches that are correctly detected in terms of a percentage of total number of branches detected as infeasible. Recall (p_d) measures a detector's ability in finding actual infeasible branches.

$$\begin{aligned} &\textit{Precision}: \; p_r = t_p/(t_p+f_p) \\ &\textit{Recall}: \quad p_d = t_p/(t_p+f_n) \end{aligned} \tag{1}$$

Efficiency: We evaluate the efficiency of an infeasible branch detector through computing its cost in time and computation. For the former, we show the average processing time for each branch (**AvgT/Br**) and the total time used for each system (**TolT/Sys**). For the latter, we showed the min (**MinP**), mean (**MeanP**), max (**MaxP**) values of the number of predicate nodes involved in symbolic evaluation per

branch. These metrics is shown in Table III. Ideally, a good detector should have low cost in both time and computational resources.

TABLE III. EFFICIENCY MEASUREMENT

Time cost	**AvgT/Br**: average processing time for each branch **TolT/Sys**: total time used for each system
Computational Cost	**#MinP**: the minimum predicates in symbolic evaluation per branch **#MeanP**: the average predicates in symbolic evaluation per branch **#MaxP**: the maximum predicates in symbolic evaluation per branch

C. *Experimental Results*

Accuracy: Table IV and Table V show the results of PIND and FullZ3 for the intra-procedural and inter-procedural analysis, respectively. The column *#Branches* stands for the total number of branches within each subject system. The column *#Seeds* stands for the number of seeded infeasible branches. The columns t_p, f_p and f_n represent the number of true-positives, false-positives and false-negatives detected by each approach respectively. The following summarizes the key results:

- Both PIND and FullZ3 do not generate any false positives for the selected seeds. All the detected infeasible branches are confirmed infeasible.
- In the intra-procedural analysis, PIND generates 89 false negative cases; FullZ3 generates 379. Therefore, over the 1,753 seeded infeasible branches, PIND has a false positive rate of 89/1,753=5.08%, which is 16.54% less than FullZ3 (379/1,753=21.62%).
- In the inter-procedural analysis, PIND generates 17 false negative cases, while FullZ3 generates 44. Therefore, over the 467 seeded infeasible branches, PIND has a false positive rate of 17/467=3.64%, which is 5.78% less than FullZ3 (44/467=9.42%).

Our experiments show that the Precision of PIND and FullZ3 are 100% for all the subject systems. Figures 9 and 10 compare the Recall values achieved by the two approaches, for intra-procedural analysis and inter-procedural analysis, respectively. We find that:

- In the intra-procedural analysis, PIND achieves higher Recall values than the FullZ3 approach. For all the subject systems, the recall improvement of PIND over FullZ3 ranges from 9.74% to 23.18%. On average, PIND achieves an average recall of 94.92%, and FullZ3 achieves an average recall of 78.38%.
- In the inter-procedural analysis, PIND also achieves higher Recall values than the FullZ3 approach. For the subject system JLibSys, both PIND and FullZ3 achieve the 100% recall. For other subject systems, the improvement of PIND over FullZ3 ranges from 3.74% to 8.16%. On average, PIND achieves an average recall of 96.36%, and FullZ3 achieves an average recall of 90.58%.

The above results show that PIND achieves high accuracy and can detect more infeasible branches than FullZ3. We will discuss the reasons of this improvement in Section V.

TABLE IV. INFEASIBLE BRANCHES DETECTION RESULTS OF INTRA-PROCEDURAL ANALYSIS

System	#Branches	#Seeds	PIND			FullZ3		
			t_p	f_p	f_n	t_p	f_p	f_n
Java								
JLibSys	5,255	151	140	0	11	105	0	46
PMD	33,108	627	594	0	33	450	0	177
JMathLib	13,452	143	136	0	7	113	0	30
Android								
Unify_Unit	23,149	503	485	0	18	436	0	67
GoLocker	10,771	183	164	0	19	141	0	42
GPSLogger	6,684	146	145	0	1	129	0	17
Grand Total	92,419	1,753	1,664	0	89	1,374	0	379

TABLE V. INFEASIBLE BRANCHES DETECTION RESULTS OF INTER-PROCEDURAL ANALYSIS

System	#Branches	#Seeds	PIND			FullZ3		
			t_p	f_p	f_n	t_p	f_p	f_n
Java								
JLibSys	816	35	35	0	0	35	0	0
PMD	4,837	107	103	0	4	99	0	8
JMathLib	4,650	120	114	0	6	109	0	11
Android								
Unify_Unit	3,307	90	85	0	5	78	0	12
GoLocker	1,795	49	49	0	0	45	0	4
GPSLogger	1,534	66	64	0	2	57	0	9
Grand Total	16,939	467	450	0	17	423	0	44

Efficiency: In Table VI and Table VII, we show the efficiency of infeasible branch detection with PIND and FullZ3. The efficiency results are measured using the metrics defined in Table III. For the intra-procedural analysis (Table VI), on average, PIND spends 17.59ms to process a branch while FullZ3 needs 26.72ms. On average, PIND spends 270.88 seconds to test a system while FullZ3 needs 411.65 seconds. Therefore PIND is (411.65-270.88)/411.65=34.20% faster than FullZ3. Furthermore, PIND processes much fewer predicates than FullZ3. On average, PIND needs to process 1.53 predicates to detect an infeasible branch while FullZ3 needs to process 2.83 predicates.

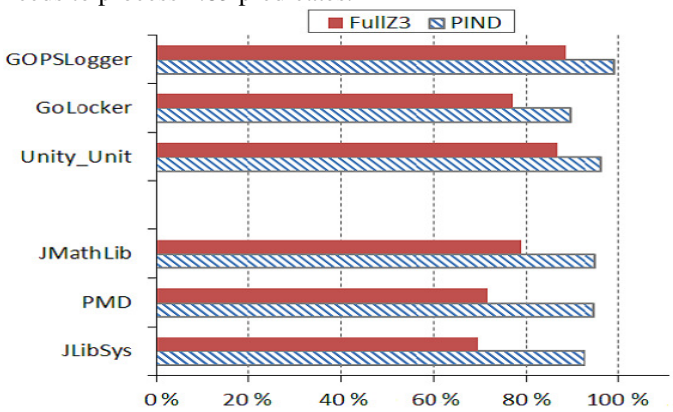

Fig. 9. The Precision results of intra-procedural analysis

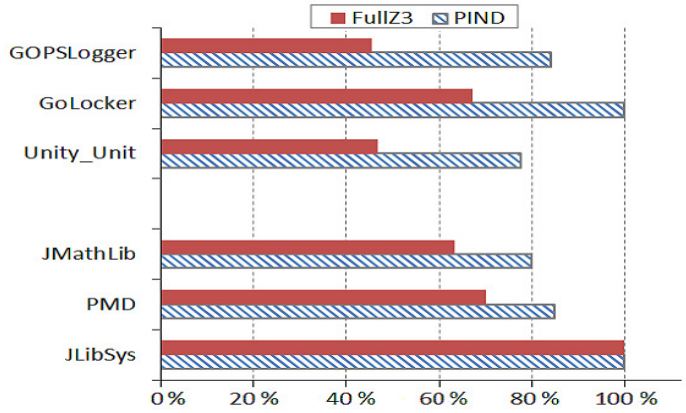

Fig. 10. The Precision results of inter-procedural analysis

For the inter-procedural analysis (Table VII), on average, PIND spends 26.50ms to process a branch while FullZ3 needs 50.89ms. On average, PIND spends 288.91 seconds to test a system while FullZ3 needs 487.42 seconds. Therefore PIND is (487.42-288.91)/487.42=40.73% faster than FullZ3. Furthermore, PIND processes much fewer predicates than FullZ3. On average, PIND needs to process 2.12 predicates to detect an infeasible branch while FullZ3 needs to process 3.19 predicates.

The above results confirm that PIND achieves higher efficiency than FullZ3.

V. Discussion of the Results

A. *Analysis of the Results*

We study the differences between the results detected by PIND and FullZ3. We summarize the findings as follows:

1) Both PIND and FullZ3 suffer the impact of nonlinear arithmetic operations and string operations. Therefore, neither of them can detect the infeasible branches that contain nonlinear arithmetic operations or string operations. In intra-procedural analysis, PIND and FullZ3 missed 89 seeded infeasible branches due to this reason. In inter-procedural analysis, PIND and FullZ3 missed 14 seeded infeasible branches due to this reason.

2) Additionally, FullZ3 generates more false negatives because it attempts to do exhaustive symbolic evaluation for every branch and therefore triggers a lot of runtime exceptions. In intra-procedural analysis, FullZ3 failed to detect 290 seeded infeasible branches due to this reason. In inter-procedural analysis, FullZ3 failed to detect 27 seeded infeasible branches due to this reason.

Figure 11 shows a code example of PMD and its CFG to demonstrate PIND's strength over FullZ3. This example contains complex nested loops and total 21 predicates. PIND uses light-weighted syntax analysis to capture a pair of predicates {node6, node8}, and uses it to detect the Semantic Dominator pattern. After symbolic evaluation, PIND confirms that the true branch of node8 is infeasible. However, when FullZ3 deals with this case, it tries to evaluate all paths in this CFG exhaustively. In our experiment, FullZ3 failed to capture the infeasible branch due to an OutOfMemoryError exception (the experiment is carried out on a PC with 4 GB memory).

```
.......
for(;;){ /*1*/
   do{
   ........
   /*2*/   if ((0xfffffdfffffffffffL & l) != 0L){
   /*3*/     if (kind > 47)
   /*4*/       kind = 47;
   /*5*/     jjCheckNAddStates(26, 28);
           }
   /*6*/   else if (curChar == 45)
   /*7*/     jjstateSet[jjnewStateCnt++] = 5;

   /*8*/   if (curChar == 45)
   /*9*/     jjstateSet[jjnewStateCnt++] = 4;
   ........./*omit another 15 predicates that are not
   correlated with node6 and node8*/
   /*10*/ }while(...)

   /*11*/ curChar = input_stream.readChar();
}
```

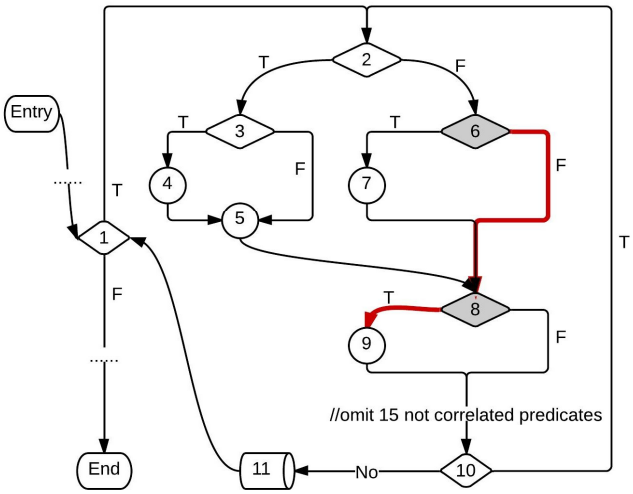

Fig. 11. A code example from PMD

B. *Application of PIND in Test Case Coverage*

Branch coverage is a commonly-used metric for evaluating test adequacy. The existence of infeasible branches suggests that the branch coverage cannot always reach 100%. Understanding the number of infeasible branches can help improve test coverage results.

We run PIND over the six systems and classify the branches into three types: infeasible, feasible and unknown. The results are shown in Tables VIII and IX. The infeasible branches take 19.64% of the total branches in intra-procedural analysis, and 25.18% of the total branches in inter-procedural analysis. Therefore, code coverage tools that do not consider infeasible branches could lead to inaccurate results.

We also carry out a case study to further verify PIND's usefulness in real-world practice. EclEmma [1] is a widely used code coverage tool for Java, but it does not consider the impact of infeasible branches. We compare the coverage analysis results between (1) solely using EclEmma and (2) using EclEmma after integrating the results of PIND.

TABLE VI. EFFICIENCY RESULTS OF INTRA-PROCEDURAL ANALYSIS

System	PIND					FullZ3				
	AvgT/Br (ms)	TolT/Sys (s)	#MinP	#MeanP	#MaxP	AvgT/Br (ms)	TolT/Sys (s)	#MinP	#MeanP	#MaxP
Java										
JLibSys	31.03	163.06	1	1.58	6	41.77	219.51	2	4.77	11
PMD	12.89	426.76	1	1.21	2	20.94	693.28	1	2.79	8
JMathLib	17.81	239.58	1	1.67	4	29.43	395.89	1	3.51	4
Android										
Unify_Unit_Converter	16.55	383.12	1	1.47	2	23.81	551.18	1	1.96	4
GoLocker	20.28	218.44	1	1.91	2	30.09	324.10	1	2.65	7
GPSLogger	29.07	194.30	1	2.36	4	42.78	285.94	1	3.47	8
Average	17.59	270.88	1	1.53	3.33	26.72	411.65	1.17	2.83	7

TABLE VII. EFFICIENCY RESULTS OF INTER-PROCEDURAL ANALYSIS

System	PIND					FullZ3				
	AvgT/Br (ms)	TolT/Sys (s)	#MinP	#MeanP	#MaxP	AvgT/Br (ms)	TolT/Sys (s)	#MinP	#MeanP	#MaxP
Java										
JLibSys	35.90	129.03	1	2.15	6	63.08	220.64	1	6.19	37
PMD	23.88	410.07	1	2.06	4	50.15	863.20	1	8.44	128
JMathLib	22.71	214.91	1	1.78	4	52.91	423.36	1	7.19	27
Android										
Unify_Unit_Converter	19.81	531.00	1	1.99	4	30.51	742.88	1	10.61	32
GoLocker	22.62	207.52	1	2.43	2	30.09	324.10	1	8.49	19
GPSLogger	34.06	240.93	1	2.29	4	70.50	361.08	1	7.07	34
Average	26.50	288.91	1	2.12	4	50.89	487.42	1	3.19	46.17

TABLE VIII. INFEASIBLE BRANCHES DETECTED BY PIND (INTRA-PROCEDURAL ANALYSIS)

System	#Branches						
	Total	#feasible	#infeasible				#un-known
			#Pattern1	%	#Pattern2	%	
Java							
JLibSys	5,255	4,555	81	1.54%	470	8.94%	149
PMD	33,108	27,891	985	2.98%	3704	11.19%	528
JMathLib	13,452	9,493	559	4.16%	2646	19.67%	754
Android							
Unify_Unitr	23,149	16,970	1542	6.66%	3968	17.14%	669
GoLocker	10,771	7,761	673	6.25%	2020	18.75%	317
GPSLogger	6,684	5,138	270	4.04%	1231	18.42%	45
GrandTotal	92,419	71,808	18,149(19.64%)				2,462

TABLE IX. INFEASIBLE BRANCHES DETECTED BY PIND (INTER-PROCEDURAL ANALYSIS)

System	#Branches						
	Total	#feasible	#infeasible				#un-known
			#Pattern1	%	#Pattern2	%	
Java							
JLibSys	816	315	105	12.87%	117	14.34%	279
PMD	4,837	2,316	823	17.01%	421	8,70%	1,277
JMathLib	4,650	2,190	571	12.28%	839	18.04%	1,050
Android							
Unify_Unitr	3,307	1,927	322	9.74%	408	12.34%	650
GoLocker	1,795	1,139	132	7.35%	207	11.53%	317
GPSLogger	1,534	1,017	80	5.22%	240	15.65%	197
GrandTotal	16,939	8,904	4,265(25.18%)				3,770

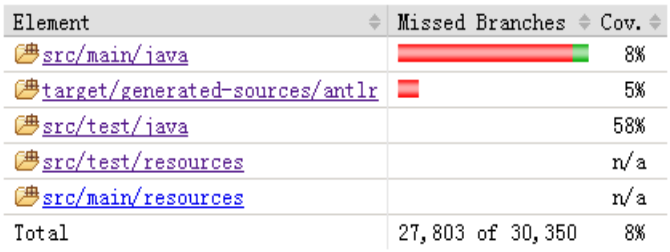

Fig. 12. EclEmma's coverage report before integrating PIND

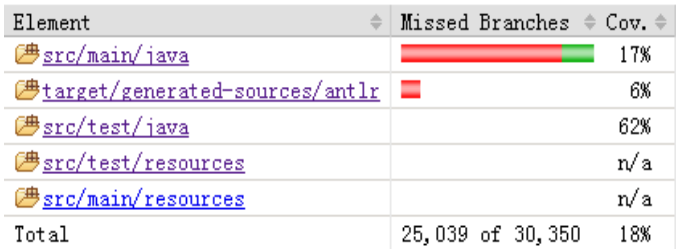

Fig. 13. EclEmma's coverage report after integrating PIND

We select the sub-project hibernate-core from Hibernate-4.3.0 [13] for this case study. Hibernate-core contains 353 test cases. The branch coverage reported by EclEmma is only 8% (intra-procedural). However such a low coverage is biased by the existence of infeasible branches. We integrate EclEmma with PIND by excluding detected infeasible branches.

Therefore, the branch coverage report is adjusted. Figure 12 and Figure 13 show the coverage reports before and after integrating PIND: with the same set of test cases, the branch coverage has been adjusted from 8% to 18%.

VI. THREATS TO VALIDITY

Although we have compared PIND with FullZ3 and carefully analyzed the results, the following issues may still pose threats to validity to our work:

- PIND instruments Z3 for constraint solving. Though Z3 is a state-of-the-art constraint solver, it still has limitations in dealing with floating point arithmetic accurately [4]. Therefore, some results may be biased due to the floating point error of Z3.
- In our evaluation, we randomly select infeasible branches and consider them as "seeds". Although we have selected many (1,753 for intra- and 467 for inter-procedural analysis) infeasible branches from the subject systems, the number of seeds is still small compared to the total number of branches (92,419 for intra- and 16,742 for inter-procedural analysis) in all the subject systems. Furthermore, we manually identify the feasibility of the selected branches and consider it as "ground truth". Therefore, the data used in our evalution may be biased.
- Although we have tested PIND on both Java and Android programs, there is still room for further improvements in terms of generality. In the future, we will experiment PIND on more types of systems including industry systems.

VII. RELATED WORK

Existing methods for detecting infeasible branches or paths can be mainly classified into three types: (a) path-sensitive method; (b) path-insensitive method and (c) code pattern based method.

A. Path-sensitive Method

Path-sensitive approaches apply symbolic evaluation to determine the infeasibility of a path/branch. These approaches carry a path sensitive analysis for each branch in a given CFG. Through symbolic evaluation, they propagate the constraint along every path on the branch and apply theorem prover to determine the solvability of the constraint during each propagation. If the constraint is always unsolvable, the branch is then concluded as infeasible. These approaches have high precision of detection but with heavy overhead. The advantage of the path-based constraint propagation approaches is the precision with which we can detect infeasible program paths. The difficulty in using full-fledged path-sensitive approaches to detect infeasible branch is the huge number of program paths to consider. In summary, even though path-based constraint propagation approaches are more accurate for infeasible branch detection, they suffer from a huge complexity.

Path-sensitive approaches are often used in areas requiring high accuracy like test case generation and code optimization. In test case generation, branches will be firstly evaluated their infeasibility. Infeasible branch will be filtered from test data generation to save resources and time. For example, Korel [14] checked the infeasibility before generating test case at the last predicate to avoid unnecessary computation. Other similar examples include the work of Tillmann and Halleux [2], and the work of Prather and Myers [15]. In code optimization, def-use pairs along those infeasible paths are eliminated for enhancing the efficiency of code [16].

B. Path-insensitive Method

Path-insensitive approaches apply general data flow analysis to determine the feasibility of a branch or a path [17, 29]. These approaches calculate and gather a list of variable-value mappings, which maps program variables to values at required locations in CFG. A node with multiple in-edges is defined as a merge location, where variable-value mappings from all of its predecessor nodes are joined together. Because of the joining operation, a variable may be mapped to a set of values instead of a single value. If a node has multiple out-edges (i.e., a predicate node), each of these out-edges is defined as a control location. Infeasibility is detected when any variable is mapped to an empty set of values at a control location [17]. However, these approaches scarify the detection precision, which may cause some infeasible paths to be wrongly identified as feasible. It is important to note that the variable-value mappings computed at a control location L is essentially an invariant property — a property that holds for every visit to L. Therefore two things could cause the loss of the detection precision: First, the correlated branches are ignored, and variable-value mappings are propagated across infeasible paths. Second, by the joining operation at merging location, variable-value mappings from different paths are joined, leading to further over-approximation [18].

C. Code Pattern Detection

In recent years, a large number of studies have been conducted to detect bugs/problems in programs based on code patterns [19-24]. These approaches utilize heuristics to optimize traditional detection approaches and have achieved promising results. The main objective of applying heuristic code patterns is replacing the full symbolic evaluation, which is expensive and error-prone, with a combination of syntax analysis and limited symbolic evaluation to achieve effectiveness.

For example, the FindBugs and PMD tools [19-21] can analyze Java code and report warnings based on hundreds of bug patterns such as null pointer dereference, read of unwritten field, etc. Livshits and Zimmermann [22] found application-specific bug patterns by mining software revision histories. Violations of these patterns are responsible for a multitude of errors. Liu et al. [23] proposed methods for testing input validation in Java-based web applications via the identification of code patterns. In our prior work [24], we focused on the domain of database applications and discovered four constraint enforcement related code patterns. We formally described the identified patterns and developed a tool to detect missing constraint enforcement based on the patterns. In [25],

several code patterns for infeasible paths in Java programs were discovered. The work presented in this paper focuses on the identification and detection of infeasible branch patterns. All these studies revealed that many types of problematic code in programs do exhibit certain patterns. Therefore, we can effectively and efficiently discover problematic code by identifying these patterns.

VIII. CONCLUSIONS

Infeasible branches are program branches that can never be exercised regardless of the inputs of the program. The existence of infeasible branches hinders program understanding and software testing. In this paper, we report our discovery of two infeasible branch patterns, namely Constant Value pattern and Semantic Dominator pattern. We propose a method PIND for automatically detecting infeasible branches based on these two patterns. The experiment results on 6 Java and Android systems show the accuracy and efficiency of PIND. On average, our tool can successfully detect the infeasible branch in the subject systems with a 100% precision in both intra- and inter-procedural analysis and a recall of 94.92% and 98.15% in intra- and inter-procedural analysis, respectively.

Some important future work is as follows:

- Test data generation: Test data can be automatically generated by exercising a selected branch in a program [26]. Together with our prior work on infeasible path detection [25], we plan to further improve current branch/path based test data generation methods.
- Bug detection: Infeasible branches are often questionable code. Studies have shown that infeasible branch lead to redundant code that may indicate possible bugs [27]. In the future, we will integrate our approach with static bug detection approaches.

ACKNOWLEDGMENT

This research is partially supported by the China NSF grants 61073006 and 91218302. We thank student Yunpeng Li for his help with the initial version of this paper.

REFERENCES

[1] EclEmma. Available: http://www.eclemma.org/, 2012
[2] N. Tillmann and J. D. Halleux, "Pex: white box test generation for .NET," Proc. of the 2nd international conference on Tests and proofs, Prato, Italy, 2008.
[3] K. Sen, D. Marinov, and G. Agha, "CUTE: a concolic unit testing engine for C," SIGSOFT Softw. Eng. Notes, vol. 30, pp. 263-272, 2005.
[4] Z3: SMT solver. Available: http://research.microsoft.com/en-us/um/redmond/projects/z3/ml/z3.html, 2013
[5] S. Sinha, M. J. Harrold, and G. Rothermel, "Interprocedural control dependence," ACM Trans. Softw. Eng. Methodol., vol. 10, pp. 209-254, 2001.
[6] E. Duesterwald, R. Gupta, and M. L. Soffa, "A practical framework for demand-driven interprocedural data flow analysis," ACM Trans. Program. Lang. Syst.,19:992-1030, 1997.
[7] L. Lu, C. Zhang, and J. Zhao, "Soot-based implementation of a demand-driven reaching definitions analysis," Proc. of the ACM SIGPLAN International Workshop on State of the Art in Java Program analysis, Beijing, China, 2012.
[8] Soot - a Java Optimization Framework. Available: http://www.sable.mcgill.ca/soot/, 2013
[9] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In Proc. of the International Conference on Software Engineering (ICSE 2012), 760-770, June 2012.
[10] A. Einarsson and J. D. Nielsen, A Survivor's Guide to Java Program Analysis with Soot. Available: http://www.brics.dk/SootGuide/, 2008.
[11] Sourceforge. Available: http://www.sourceforge.net/, 2013
[12] T. McCabe, A complexity measure. IEEE Transactions on Software Engineering, 2(4):308–320, Dec 1976.
[13] Hibernate. Available: http://planet.jboss.org/post/hibernate_orm_4_3_0_beta1_release, 2013
[14] B. Korel, "Automated test data generation for programs with procedures," SIGSOFT Softw. Eng. Notes, vol. 21, pp. 209-215, 1996.
[15] R. E. Prather and J. J. P. Myers, "The Path Prefix Software Testing Strategy," IEEE Trans. Softw. Eng., vol. 13, pp. 761-766, 1987.
[16] R. Bodic, R. Gupta, and M. L. Soffa, "Refining data flow information using infeasible paths," SIGSOFT Softw. Eng. Notes, vol. 22, pp. 361-377, 1997.
[17] U. Khedker, A. Sanyal, and B. Karkare, Data Flow Analysis: Theory and Practice: CRC Press, Inc., 2009.
[18] J. Fischer, R. Jhala, and R. Majumdar, "Joining dataflow with predicates," SIGSOFT Softw. Eng. Notes, vol. 30, pp. 227-236, 2005.
[19] FindBugs. Available: http://findbugs.sourceforge.net, 2012.
[20] D. Hovemeyer and W. Pugh, "Finding bugs is easy," SIGPLAN Not., vol. 39, pp. 92-106, 2004.
[21] PMD. Available: http://pmd.sourceforge.net/, 2013.
[22] B. Livshits and T. Zimmermann, "DynaMine: finding common error patterns by mining software revision histories," SIGSOFT Softw. Eng. Notes, vol. 30, pp. 296-305, 2005.
[23] H. Liu and H. B. K. Tan, "Testing input validation in Web applications through automated model recovery," J. Syst. Softw., vol. 81, pp. 222-233, 2008.
[24] H. Zhang, H. B. K. Tan, L. Zhang, X. Lin, X. Wang, C. Zhang, and H. Mei, "Checking enforcement of integrity constraints in database applications based on code patterns," J. Syst. Softw., vol. 84, pp. 2253-2264, 2011.
[25] M. N. Ngo and H. B. K. Tan, "Heuristics-based infeasible path detection for dynamic test data generation," Inf. Softw. Technol., vol. 50, pp. 641-655, 2008.
[26] C. Hu and I. Neamtiu, "Automating GUI testing for Android applications," Proc. of the 6th International Workshop on Automation of Software Test, Waikiki, Honolulu, 2011.
[27] Y. Xie and D. Engler, "Using redundancies to find errors," SIGSOFT Softw. Eng. Notes, vol. 27, pp. 51-60, 2002.
[28] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," ACM Comput. Surv., vol. 29, pp. 366-427, 1997.
[29] J. Gustafsson, A. Ermedahl, C. Sandberg, and B. Lisper, "Automatic Derivation of Loop Bounds and Infeasible Paths for WCET Analysis Using Abstract Execution," Proc. of the 27th IEEE Int' l Real-Time Systems Symposium, 2006, pp.57-66.