Journal of Software Engineering

ISSN 1819-4311

Academic Journals Inc.

www.academicjournals.com

Journal of Software Engineering 6 (4): 65-78, 2012
ISSN 1819-4311 / DOI: 10.3923/jse.2012.65.78

Detecting Infeasible Paths via Mining Branch Correlations

[1]Cheng Zhang and [2]Yuting Chen
[1]Department of Computer Science and Engineering, Shanghai Jiao Tong University, China
[2]School of Software, Shanghai Jiao Tong University 800 Dongchuan Road, Shanghai 200240, China

Corresponding Author: Cheng Zhang, Department of Computer Science and Engineering, Shanghai Jiao Tong University, 800 Dongchuan Road Shanghai 200240, China Tel: +86-21-34204043 Fax: +86-21-34205145

ABSTRACT

The existence of infeasible program paths remains an obstacle in applying static analysis to software engineering activities, such as test data generation and bug finding. Knowledge about these infeasible paths is valuable to improve the precision of static analysis. This study presented a hybrid approach for detecting infeasible paths effectively, by combining program analysis and data mining techniques. The approach is based upon two assumptions: (1) most infeasible paths are caused by branch correlations and (2) runtime values of correlated branch predicates can display certain patterns which reveal the underlying correlations between the branches. The approach discovers the possible correlation rules by mining the data collected through instrumenting target programs and executing adequate test cases. Then the approach scores the infeasibility of each program path according to the number of rules it breaks. The evaluation shows that the approach can detect a large portion of infeasible paths and some of them are difficult to be identified by existing infeasible path detecting methods.

Key words: Infeasible path, program analysis, branch correlation, data mining, association rules

INTRODUCTION

Static analysis is the basis of many software engineering methods and tools. It has been used to support code optimization (Aho *et al*., 1986), test data generation (Gupta *et al*., 2000; Jasper *et al*., 1994; Ngo and Tan, 2008), performance analysis (Dufour *et al*., 2007), bug finding (Hovemeyer and Pugh, 2004) and many other activities. A common assumption made in static analysis is that every program path is executable. But it may be overly conservative, as there exist a notable amount of infeasible paths which can never be exercised during program execution. This conservative assumption is acceptable in code optimization, as the compiler should give up possible chances for optimization rather than risk the correctness of program. However, when it comes to other realms, such as bug finding, it may bring intolerable drawbacks, one of which is the high false positive rate in bug reports (Hovemeyer and Pugh, 2007).

Although, it is not viable to generally solve the problem of detecting infeasible paths (Hedley and Hennell, 1985), a variety of partial solutions have been proposed. Static techniques (Bodik *et al*., 1997b; Goldberg *et al*., 1994; Malevris, 1995; Zhang and Wang, 2001) typically rely on symbolic execution, control flow and data flow analysis. They are powerful enough to find infeasible paths caused by correlations between conditional predicates of simple forms, such as individual boolean variables or numerical comparisons. However, these static techniques may not work well with complicated conditional predicates, especially those containing function calls.

Dynamic techniques like (Bueno and Jino, 2000) often monitor the values of several variables and the path directions, using certain strategies or algorithms to direct path selection and test data generation. The most recent heuristics-based methods (Ngo and Tan, 2007, 2008) are able to detect a considerable amount of infeasible paths by identifying code patterns and correlations between branches. These empirical approaches proved to be quite precise and efficient. However, to the best of our knowledge, there are still some special kinds of infeasible paths that are difficult to detect with existing techniques. Here is an example written in Java:

```
• If(node.getParent() == doc) // p1
• Print("Doc is node's parent."); // stmt1
• If(doc.getChildren().size() > 0) // p2
• Print("Doc has some children."); // stmt2
```

where node.getParent()returns the parent of node and doc.getChildren() returns a list of children of doc. According to this specification, whenever node.getParent() == doc evaluates to true, doc.getChildren().size()>0 also evaluates to true. Thus, a program path that goes through stmt1 and skips stmt2 can never be executed. This infeasible path is caused by the correlation between p1 and p2. It is worth noting that such kind of correlation between predicates containing method calls may not be easily discovered by techniques based on static analysis, because structures and semantics of the methods may be too complex to analyze or evaluate statically. In addition, the polymorphism issues may further complicate the analysis for object-oriented programs. As p1 and p2 contain different sets of variables, it is also challenging for empirical approaches like (Ngo and Tan, 2008) to identify such correlations. Furthermore, this example falls into none of the four code patterns summarized (Ngo and Tan, 2007).

The objective of this study is to propose an infeasible path detecting approach that can overcome the aforementioned problems of existing techniques and is able to detect a large portion of infeasible paths effectively.

BACKGROUND

Infeasible path and branch correlation: An infeasible path is a path which cannot be exercised by any set of input values. Infeasible paths are contained in almost every real program and infeasible path detection is the job to detect the possible infeasible paths in the program. Because of the existence of infeasible paths, considerable efforts may be wasted in test data generation and, more seriously, imprecise results may be generated by bug finding tools like FindBugs (Hovemeyer and Pugh, 2004). Most static bug finding tools make the assumption that every program path is executable, though some of them have put effort on infeasible path detection. Our approach analyzes the branch correlations in the program by mining dynamic data, in order to improve the efficacy of infeasible path detection.

Branch correlation is the relationship among conditional branches (Bodik *et al*., 1997a; Mueller and Whalley, 1992). There is a branch correlation when the outcome of a conditional branch is determined by the previous branches. Branch correlations are represented in various formats. For example, suppose there are two conditional branches in a program are p1 and p2. If there is the relation that when the output value of p1 is true, the output value of p2 is definitely true, then a branch correlation exists between p1 and p2. As a result, a path must be infeasible

when it goes through, both p1 and p2 with the condition that the output value of p1 is true while the output value of p2 is false. Although, the example is simple, it shows how program paths become infeasible due to branch correlations.

Basics of association rule mining: Association analysis (Tan *et al*., 2006) is a data mining technique that discovers association rules in large data sets. An association rule is an expression X→Y, where X and Y are disjoint item sets. For a data set D where a rule X→Y can be found, this rule indicates that any item set that contains X may contain Y as well with a given probability. This conditional probability, or rule confidence, along with rule support, are two measurements to evaluate the strength of an association rule. Specifically, the formal definition of these two metrics is:

$$\text{Support, } s(X \rightarrow Y) = p\,(XUY) \tag{1}$$

$$\text{Confidence, } c(X \rightarrow Y) = p\,(Y|X) \tag{2}$$

Intuitively, a rule of low support is unfavorable because such rule may simply occur by chance which indicates that this rule is very likely to be spurious. Similarly, a rule that fails to satisfy the threshold of confidence is also undesirable since with a relatively low reliability of the inference, such a rule is trivial and cannot be used to substantiate a co-occurrence relationship between two item sets.

One common application domain of association analysis is to extract patterns in market transaction records which can help retailers use these patterns to promote cross-selling of their products. To apply association analysis, our research first collects execution information of branch predicates in the program and then identifies the branch outcomes that appear together frequently and form association rules based on these frequent "item sets". However, there are two main limitations in association analysis. One limitation is that the overhead in both terms of time and space often makes analyzing large-scale programs impractical. Another limitation is that patterns discovered are potentially spurious, thus generating rules that contain no correlation relationship between two sets of predicates.

To handle the first limitations, our approach have optimized the association rule mining in Weka (Hall *et al*., 2009; Witten and Frank, 2005), by adapting the data structures and algorithms for our specific purpose. As a result, the time consumption has been reduced by up to 50%. In addition, different execution instances of the same branch predicate inside a loop are not placed in one transaction, because it will greatly increase the number of items. Instead, they are separated into smaller transactions while still enable identifying correlation relationship from different loops.

To handle the problem of spurious rules, our approach carefully chooses the values of support and confidence, more specifically; minimal confidence is set to 1.0 for generating rules that only contains strong correlation relationship.

EXAMPLE

This section illustrates the presented approach by applying it to the example Java program shown in Fig. 1. In the program, there are five branch predicates, each followed with a comment showing its alias pi, i = 1, 2, 3, 4, 5. The Control Flow Graph (CFG) (Harrold *et al*., 1993) of method ipExample is shown in Fig. 2, in which each node is labeled with the corresponding line number, except for two special nodes, namely nodes S and E. The node S is the CFG entry and the node E is the CFG exit.

```
public void ipExample (Product p, Customer c){
1. if(p.isExpensive()) //p1
2. record customer(c);
3. if(p.is Discountable()) //p2
4. discount(p);
5. boolean flag = false;
6. int index = 0;
7. while(flag == false){ //p3
8. if(pIDs[index] == p.getID()) //p4
9. flag = true;
 else
10. index++;
 }
11. if(p.isDiscountable()) //p5
12. pStates[index] = SOLD_WITH_DISCOUNT;
 else
13. pStates[index] = SOLD;
}
```

Fig. 1: An example of program

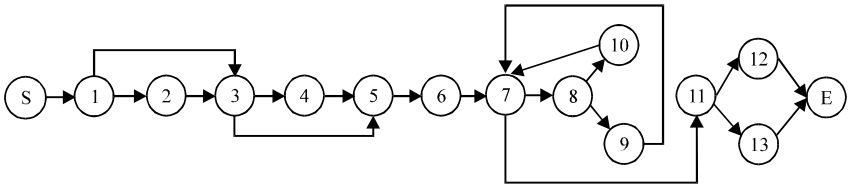

Fig. 2: CFG of method ipExample

There are four branch correlations in the program. First, a business rule specifies that expensive products will never be discounted. As a result, p1 and p2 are mutually exclusive, that is, ((p1 == true)↔(p2 ==false)) and ↔ ((p1 == false) ↔ (p2 == true). Second, the while loop is used to locate the index of a given product in an array. When the product IDs match, the flag flips and the loop terminates. Therefore, the correlation between p3 and p4 is (p4 ==true) →(p3'==false), where p3' represents the appearance of p3 right after p4 in the loop. Finally the state of product is set according to whether the product is discountable, which induces two more correlations: p1 and p5 are mutually exclusive, while p2 and p5 are identical.

In the first phase, our approach instruments the Java bytecode of this program to monitor runtime evaluations of the five branch predicates. After executing the program with a number of test cases, our approach collects several execution traces in the form of sequences of values of the predicates, such as {p1 == true; p2 == false; p3 == true; p4 == true; p3' == false; p5 == false}. In the second phase, our approach uses association rule mining to analyze the execution data in order to discover the correlations between the branch predicates. In the example shown in Fig. 1, there are over 100 rules found, but only 14 of them (shown in Table 1) are considered as valid after a series of pruning procedures. In the third phase, our approach scores each path by finding out how many rules it breaks. For example, the path <S, 1, 3, 5, 6, 7, 8, 10, 7, 11, 12, E> breaks three rules, namely rules 1, 6 and 10. Thus it gets a high score and is identified as a highly suspicious infeasible path.

Table 1: Predicate correlations in ipExample

Association rule	Confidence	Support
p4 = = false→p3' = = true	1	2044
p1 = = true→p2 = = false	1	208
p3 = = false→p4 = = No	1	200
p3 = = false→p3' = = No	1	200
p4 = = true→p3' = = false	1	200
p1 = = false→p2 = = true	1	192
p1 = = true→p3 = = true	1	208
p1 = = true→p5 = = false	1	208
p2 = = false→p3 = = true	1	208
p2 = = false→p5 = = false	1	208
p1 = = false→p3 = = true	1	192
p1 = = false→p5 = = true	1	192
p2 == true→p3 = = true	1	192
p2 == true→p5 = = true	1	192

APPROACH TO INFEASIBLE PATH DETECTION

This section describes our infeasible path detecting approach in detail. The approach consists of three main phases: collecting data through code instrumentation, discovering branch correlations using association rule mining and detecting infeasible paths by scoring path infeasibility.

Collecting data: In the first phase of our approach, execution data of the program is collected. Our approach includes the implementation of an instrumentation framework to collect predicate jumping information, which is built on the top of ASM, a Java bytecode manipulation framework. During the runtime monitoring, every branch predicate is assigned one of the possible values, which include No, Undef, true and false. In every method, a lot of dynamic traces are collected and a preprocess step transforms the traces into simple traces. A simple trace is a trace where each predicate, except the predicates of loop entries, appears only once. A predicate of a loop entry may appear twice in a simple trace, because the second value of the predicate may have correlations with the predicates in the loop body.

In the program shown in Fig. 1, if there are two dynamic traces collected, namely t1 := {p1 == true; p2 == true; p3 == false; p5 == true} and t2 := {p1 == true; p2 == true; p3 == true; p4 == false; p3 == true; p4 == true; p3 == false; p5 == true}, several simple traces will be computed in the following way.

Our approach first splits the loop and adds some predicates with value No if these predicates exist in the program but are not executed. In the above example, this step will add p4 and p3' with value No in the t1 trace. Then a trace that contains a loop will be separated into several simple traces. Every simple trace contains one loop body of the original trace and the predicate of the loop entry will appear twice. In the above example, t2 contains three iterations of the loop. The first iteration is {p3 == true; p4 == false; p3' == true}, the second iteration is {p3 == true; p4 == true; p3'== false} and the last iteration is {p3 == false}. Here the predicate p3' means the next value of p3. The first simple trace contains the predicates before the loop and other predicates are assigned Undef, because the predicates before the loop may directly affect the first iteration of loop. Therefore, the first simple trace from t2 is {p1 == true; p2 == true; p3 == true; p4 == false; p3' == true; p5 == Undef}. Similarly the last simple trace contains the predicates after the loop and

other predicates are assigned Undef, due to the fact that the last iteration of the loop body may directly affect the predicates after the loop. The last simple trace from t2 is {p1 == Undef; p2 == Undef; p3 == false; p4 == No; p3' == No; p5 == true}. In other simple traces, our approach just sets all the predicates before and after the loop to Undef. For example, the second simple trace of t2 is {p1 == Undef; p2 == Undef; p3 == true; p4 == true; p3' == false; p5 == Undef}. There is a special simple trace in which all the predicates in the loop body are set to Undef. It contains the values of the predicates before and after the loop body so that correlations among the predicates before and after the loop can be captured. The special simple trace for t2 is {p1 == true; p2 == true; p3 == Undef; p4 == Undef; p5 == true}.

Finally, the collected dynamic data is represented in the form of simple traces which will be used to mine correlations between predicates. The final simple traces for t2 include:

- {p1 == true; p2 == true; p3 == false; p4 == No; p3' == No; p5 == true}
- {p1 == true; p2 == true; p3 == true; p4 == false; p3' == true; p5 == Undef}
- {p1 == Undef; p2 == Undef; p3 == true; p4 == true; p3' == false; p5 == Undef}
- {p1 == Undef; p2 == Undef; p3 == false; p4 ==No; p3' == No; p5 == true}
- {p1 == true; p2 == true; p3 == Undef; p4 == Undef; p5 == true}

Mining branch correlations: In the second phase, our approach performs association rule mining on the collected execution data by using an open source data mining tool, Weka. Weka has implemented the classical algorithms for association rule mining that can be used directly. However, there is an inherent problem in association analysis. It may generate many spurious association rules that are trivial or even misleading. In order to solve this problem, our approach performs a series of procedures to prune the rules generated by Weka.

The first procedure is to restrict the form of rules. This procedure only retains a rule that satisfies the following conditions:

- The premise of the rule contains no more than two predicate-value pairs
- The consequence of the rule contains exactly one predicate-value pair

The rationale behind these criteria is that real branch correlations seldom involve more than three predicates and a rule whose consequence contains more than one predicate-value pairs usually can be subsumed by other rules whose consequences contain only one predicate-value pair. The second procedure is to eliminate rules containing Undef values. There are several execution traces containing predicate-value pairs like pi == Undef. However, as the Undef value is used to represent a value of predicate that should not be considered due to the execution of loops, it does not make sense when it appears in an association rule for representing a branch correlation. Unfortunately, the data mining algorithms treat Undef as a normal value and generate rules containing it. Thus these useless rules have to be filtered out after they are generated. In addition, our approach recalculates the confidence of the remaining rules after the elimination of the Undef value. Consider a case where there are 50 execution traces containing p1 == false, p2 == Undef and another 50 traces containing p1 == false, p2 == true. Before the second procedure is performed, both the rules ((p1 == false)→(p2 == Undef)) and ((p1 == false)→(p2 == true)) have a confidence of 0.5. However, the confidence of the rule ((p1 == false)→(p2 == true)) should be 1.0, if there is no effect of the Undef value, because p1 == false actually implies p2 == true according to the original execution data. In this procedure, confidence values are recalculated with the formula below:

$$c'((pi == v_1) \rightarrow (pj == v_2)) = \frac{c((pi == v_1) \rightarrow (pj == v_2))}{1 - c((pi == v_1) \rightarrow (pj == Undef))} \quad (3)$$

where c' ((pi==v_1)→ ((pj==v_2) is the new confidence of the rule ((pi==v_1)→ (pj==v_2)), is the original confidence and c ((pi==v_1)→((pj==Undef)) is the confidence of the rule ((pi==v_1)→(pj==Undef)). Note that neither v_1 nor v_2 is Undef.

The third procedure is to set the lower bounds of confidence and support. The lower bound of confidence is set to 1.0, which means the branch correlation rules whose confidences are lower than 1.0 will be eliminated. The reason is that any counter example in dynamic data can disprove a rule no matter how high its confidence is. For example, there are 99 execution traces where p1==true and p2==false and there is only one execution trace where p1==true and p2==true. The confidence of the rule ((p1==true)→(p2==false)) is 0.99, but the rule is considered invalid. Compared with the lower bound of confidence, that of support is not straightforward to decide. If the lower bound of support is too low, many coincident rules may be retained. Otherwise, if the lower bound of support is too high, valid rules may be ignored. In our experiment, a lower bound of support that is set to a value between 0.01 and 0.05 can generally result in an appropriate rule set.

The fourth procedure is to eliminate rules that are not in the evaluation order. A rule is not in the evaluation order means that its premise contains some predicates that are evaluated after the evaluation of the predicate in its consequence. For example, in Fig. 1, p1 is always evaluated before p2 during execution. Consequently, rules in the form of ((p2==v_1)→(p1==v_2)) will be eliminated by the fourth procedure. The reason for this procedure is that correlation rules should indicate the actual cause and effect relations between predicates in runtime. In Fig. 1, the evaluation of p1 may affect that of p2 but not vice versa. Moreover, most typical correlations, namely mutual exclusion and identity, will be represented by rules along with their converse-negative counterparts. For example, as p1 and p2 are mutually exclusive, Weka will probably discover two rules ((p2==true) → (p1==false)) and ((p2==false)→(p1==true)), together with two more rules ((p1==true)→(p2==false)) and ((p1==false)→(p2==true)). The latter two rules are logically equivalent to the former ones and in the evaluation order (i.e., p1 is always evaluated prior to p2). As a result, the fourth procedure can eliminate the former two rules without losing correlation information.

The fifth procedure is to eliminate rules that can be implied by others. In this setting, rule r_1 is implied by rule r_2 as long as the following conditions are satisfied:

- The set of predicate-value pairs in the premise of r_1 is a superset of that in the premise of r_2
- The consequences of r_1 and r_2 are identical

For example, if r_1 is ((p1==true, p2==false)→(p5==false)) and r_2 is ((p1==true)→(p5==false)), then r_1 is considered to be implied by r_2, which is denoted as $r_2 \rightarrow r_1$. Such an implication is usually caused by a very frequent predicate evaluation, like p2==false in this example. Most of the time, rules that can be implied by others add no valuable correlation information and can be eliminated safely.

Through the rule pruning procedures, our approach can reduce the rules generated by Weka significantly. In our experiments on programs containing about 10 branch predicates, there are usually less than 20 rules left, whereas Weka originally generates hundreds of them.

Detecting infeasible path: In the third phase, with the pruned set of branch correlation rules, our approach detects likely infeasible paths by finding out how many rules they break.

Given a program path π, our approach first determines the corresponding evaluation of each branch predicate contained in π. For example, suppose ð is the path (S, 1, 3, 5, 6, 7, 8, 10, 7, 11, 12, E) in Fig. 2. In order to execute π, the branch predicates should be evaluated as {p1==false, p2 == false, p3==true, p4==false, p3'==false, p5==true}, where p3'==false means that predicate p3 is evaluated to false in the second loop iteration. Our approach can find out all the possible targets of every branch jump by statically analyzing the source code (or Java bytecode level instruction). Thus it is straightforward to determine the evaluation of a predicate by checking which statement is executed right after it.

The key step to detect an infeasible path is to check its corresponding predicate evaluation against the branch correlation rules. Suppose π is a program path and its predicate evaluation, E(π), is {p1==v_1, p2== v_2,...,pi== v_i,...,pn== v_n}. Then π is said to break a rule r if:

- Predicate-value pairs contained in the premise of r must also be contained in E(π)
- The predicate in the consequence of r must have a value different from that in E(π)

For instance, the path (S, 1, 3, 5, 6, 7, 8, 10, 7, 11, 12, E) breaks the rule ((p1==false)→(p2==true)), because the premise of the rule, p1==false, is also contained in the predicate evaluation of the path and predicate p2 has a value true in the consequence of the rule which is inconsistent with the evaluation p2==false. A program path is checked against all the branch correlation rules and identified as a likely infeasible path if it breaks any rule.

However, as the branch correlation rules are generated by mining execution data, they are probably inadequate to represent the complete runtime behavior of the program. To remedy this limitation to some extent, our approach differentiates the detected likely infeasible paths with their rule breaking scores. The rule breaking score of a path π, RBS(π), is defined by the equation below:

$$RBS(\pi)=\sum_{r\in RB(\pi)} s(r) \quad (4)$$

Here RB(π) is the set of rules that are broken by π and s(r) is the support of the rule r. A rule with a higher support is more likely to reveal a true branch correlation in the program. Therefore, a path with higher rule breaking score is considered to have higher probability to be infeasible. Such prioritization can be useful, when too many paths identified as infeasible because of the lack of representative dynamic data.

EVALUATION

Experiment design: This study conducted an experiment to show the effectiveness of our approach in detecting infeasible paths. To evaluate our approach with better accuracy and generality, this study selectively chose some programs written in Java from Software-artifact Infrastructure Repository (SIR) (Do *et al.*, 2005). These are real-world programs used in various application domains. More importantly, SIR provides comprehensive test suites for these programs. Siena is an Internet-scale event notification middleware. JTopas is a Java library used for parsing text data. XML-security is a component library implementing XML signature and encryption standards. NanoXML is a small XML parser for Java. Table 2 shows the basic information of the subject programs used in our experiment.

Table 2: Information of subject programs

Program	LOC	Classes	Chosen methods
JTopas	5400	50	37
Siena	6035	26	18
NanoXML	7646	24	28
XML-security	16800	143	48

In order to perform the experiment, this study implemented all the three phases of our approach. To instrument on bytecode level, the implementation identified the instruction type and instrumented on conditional jump instructions with the assistance of ASM. The implementation also instrumented at the entries and exits of each method to determine the code area of the method. A splitting algorithm was implemented to process the traces to create the simple traces for data mining. The original correlations were obtained by using the classical Apriori algorithm provided by Weka. In order to obtain the path information of programs, the implementation used the control flow graphs provided by FindBugs which can build control flow graphs by analyzing bytecode without losing predicate information. The scoring system was built referring to Eq. 4 based on the information of branch correlations and program paths.

It is worth noting that the experiment used methods with two or more predicates, because no branch correlation will be shown if a method with only one predicate is mined. The chosen methods were also equipped with at least 100 test cases so that the dynamic data could be reasonably representative.

To accurately evaluate the result, the precision and recall were calculated in the experiment. To this end, this study defined some variables, including N_{real}, the number of real infeasible paths contained in the program, N_{right}, the number of real infeasible paths detected by our approach and $N_{detected}$, the number of infeasible paths (including false positives) in the program detected by our approach. Precision reflects the correctness of the detected infeasible paths and can be computed as follows:

$$\text{Precision}(\%) = \frac{N_{right}}{N_{detected}} \times 100 \tag{5}$$

Recall reflects the coverage of the infeasible path detected and can be computed as follows:

$$\text{Recall}(\%) = \frac{N_{right}}{N_{detected}} \times 100 \tag{6}$$

In the experiment, the control flow graphs of each program were manually analyzed to identify real infeasible paths which were used as ground truth to compute the values of precision and recall.

RESULTS

The results of our experiment are shown in Table 3 which consists of six columns. The first column shows the program names, the next three columns show the values of N_{real}, N_{right} and $N_{detected}$ and the last two columns show the values of precision and recall.

Table 3: Experimental results of infeasible path within four programmes

Program	N_{real}	$N_{detected}$	N_{right}	Precision	Recall
Siena	20	57	20	0.35	1.00
JTopas	31	76	29	0.38	0.94
XML-security	130	533	116	0.22	0.89
NanoXML	10	53	6	0.11	0.60
Total	191	649	171	0.26	0.90

As shown in Table 3, there were totally 191 infeasible paths in the chosen methods of these four subject programs. Our approach detected 649 infeasible paths with 171 of them were correct. In other words, 26.3% of the detected paths were truly infeasible and these paths counted for 89.5% of all the infeasible paths in the programs.

The value of precision was relatively low, which means that many false positives were generated by our approach. The accuracy of our approach is highly sensitive to the test data of the programs. The branch coverage of the test suites has great influence on the data mining results. By reviewing false positives, this study found that some feasible paths were identified as infeasible because they were not covered by test data. Lacking boundary test cases, some conditions of a predicate were never reached and paths through these conditions were likely to be detected as infeasible. With the improvement of test suites, our approach may eliminate a large number of incorrectly detected infeasible paths so that the false positive rate can be lowered significantly. Another cause of false positives was the wrong judgment on some infeasible paths. Manually detecting infeasible paths was a laborious task, so it was possible for us to miss some real infeasible paths, which might mislead our calculation of the result. In particular, some paths considered as false positives were actually infeasible when taking into account inter-procedural infeasibility. In general, the above two problems were the major causes of the low precision value.

As shown in Table 3, the value of recall reflected that some infeasible paths were not well detected by our approach. Again the main cause was the absence of adequate test cases. Some branch correlations cannot be successfully detected because only branch correlations with sufficient support were retained in the mining results. Another cause was that some branch information might be skewed due to the bytecode level instrumentation. Original branch statements in program source code might be transformed into a group of conditional jumps in bytecode, which occasionally caused difficulties in accurate instrumentation.

By reviewing the results in detail, this study found some interesting infeasible paths. Fig. 3 lists a special infeasible path detected by our approach. It is a code snippet excerpted from NanoXML. The predicates of interest are p1 and p2 in the processEntity method. It is easy for a static analysis approach to find the correlation ((p1==false)→(p2==false)) and so for our approach. However, another less obvious branch correlation, ((p1==true)→(p2==false)), can also be detected by our approach. The correlation results in another infeasible path. If p1==true, which means the entitySet does not contain the specific key, the key will be put into the entitySet. When it comes to p2, the specific key is already in the entitySet, so p2 is always false given p1==true. The change on entitySet is likely to be neglected by approaches based on empirical assertions, like (Ngo and Tan, 2008). Our approach can find branch correlations without concerning specific programming patterns. This infeasible path is interesting, because it is hard for approaches based on static analysis and empirical rules to detect. In particular, there is a function call between two predicates and the variables affecting the values of these two predicates have been changed. This

```
protected void processEntity(IXMLReader reader)
throws IOException{...
1. switch(ch){
 ...
2. case "":
3. case '\":
 ...
4. if (!entitySet.containsKey(key)) { //p1
5. entitySet.put(key,
 this.scanString(reader)); //stmt1
 }
 ...
6. break;
7. default:
8. this.skipTag(reader);
 }
9. if ( (systemID != null) &&
(! entitySet.containsKey(key)) ) { //p2
 ...
 }
...}
```

Fig. 3: Code snippet with a special infeasible path

change cannot easily be detected by static analysis since dealing with function call involves inter-procedure analysis. Moreover, even if some static approaches are sensitive to the change of variables, it is still not easy because the method put in entitySet might be very complex and thus it can be costly to judge the change. Also empirical approaches might not deal with this case more effectively. Ngo and Tan (2008) reported that a similar code snippet caused a false positive in their experiment. These two predicates were viewed as e-correlated by their approach and they ran a single test case to determine mutually exclusive or equivalent but these two predicates did not hold a relation, neither mutually exclusive nor equivalent.

Cost analysis: In the experiment, the size of a class file was typically doubled after being instrumented. Although it is acceptable for small subject programs, it may take large disk space when applied to larger programs. The cost could be reduced by using more sophisticated instrumentation algorithms.

Our experiment was conducted on a workstation with a 3.0 GHz CPU and 1.0 GB memory. The whole process of experiment, including instrumentation, test case execution, association rule mining and infeasible path detecting, took over 12 h. For some methods that contain more than 30 predicates, the hosting JVM terminated with an out of heap space error, which seemed unacceptable for such small programs. Nevertheless, the time and space cost could be dramatically reduced by optimizing the data mining algorithms.

DISCUSSION

This study is largely motivated by the heuristics-based approach proposed by Ngo and Tan (2008). In previous work Ngo and Tan (2007), summarized four code patterns that characterize a

large portion of infeasible paths. Ngo and Tan (2008) introduced the concept of empirical correlation. Two conditional statements are e-correlated if they satisfy some conditions on control flow dependence and the variables contained in them. The relation between two e-correlated predicates can be either mutually exclusive or identical. Dynamic information was used to determine their relation. Their e-correlation based approach was statistically justified and proved to be effective in infeasible path detection. Different from their empirical approach, our approach leverages data mining techniques to discover branch correlations, while theirs used static information to identify e-correlations. Because some conditional statements are not easily found to be e-correlated, our approach can detect certain types of infeasible path that cannot be identified by their approach.

Bodik *et al.* (1997b) presented an infeasible path detecting approach and used it to improve the precision of traditional def-use pair analysis. Their approach detected branch correlations by resolving predicate expressions backwards in the control flow graph during compile time. Infeasible paths were detected with a forward propagation algorithm and marked in the control flow graph. Similar to their work, our approach is also based on the assertion that branch correlations may give rise to infeasible paths. However, branch correlations involving complex predicate expressions may not be detected by their work, because they may not analyzable due to limitations of symbolic evaluation and static analysis. Compared with their work, our approach uses runtime evaluation of predicate expressions, making it insensitive to the structure of expressions.

Symbolic evaluation and theorem proving were used to test the feasibility of program paths in (Goldberg *et al.*, 1994; Jasper *et al.*, 1994). In their approaches, formulas in first order logic were constructed for control flow paths via symbolic evaluation and a theorem prover was used to test the satisfiability of these formulas. The paths were feasible if and only if the formulas were satisfiable. Zhang and Wang (2001) and Zhang *et al.* (2004) approach for infeasible path detecting was based on combination of symbolic execution and constraint solving. A set of constraints were generated from a given path and solved by a constraint solver to determine the feasibility of the path. Such kind of formal approaches are generally rigid and precise. However, they may have limitations on the scale of programs to be tested or the data structures of variables involved. Visser *et al.* (2004) presented an approach based on symbolic execution and model checking. Their approach scales well, but can only deal with linear integer constraints. In general, as our approach discovers branch correlations by mining dynamic data, it does not suffer the limitations of those based on symbolic evaluation and theorem proving.

Bueno and Jino (2000) identified the infeasibility of paths by monitoring the search progress of genetic algorithms. They introduced a new fitness function that combined control and data flow dynamic information and used it to verify the strong correlation between the lack of search progress and the infeasibility of the path. Their work is similar to our approach in that both techniques collect runtime information via code instrumentation and detect infeasible paths based on heuristics. However, our approach only records runtime evaluations of branch predicates rather than data flow information. Moreover, our basic heuristics is about branch correlation which is different from the fitness function related heuristics used in their work.

Malevris (1995) proposed a path generation method for testing where the infeasibility of paths was predicted based on the assertion that the more branch predicates a program contains, the more it is likely to be infeasible. Yates and Malevris (1989) also gave a statistical justification of this assertion. Bertolino and Marre (1994) presented an algorithm based on control flow analysis to find a path cover during branch testing. Their algorithm chose the next unconstrained arc in a reduced

flow graph, minimizing the number of predicates in each path. In this way, it derived the most likely feasible path cover. Although these works and ours all focus on conditional statements in programs, our work discovers correlations between predicates, while the other works mainly concern about the number of predicates.

CONCLUSION

This study presented a hybrid approach to detecting infeasible path. The main contributions of this study include:

- A novel approach to detect infeasible paths by mining association rules of branch predicates. Program paths breaking these rules are identified as suspicious infeasible paths. In addition, support values of the rules are used to prioritize the detected infeasible paths, which can be useful when there are too many paths found infeasible
- An implementation of prototype to evaluate the effectiveness of our approach. The prototype consists of a code instrument framework, a trace analyzer, a data mining component and a scoring system
- An evaluation has been conducted to show that our approach can achieve a high recall and detect some infeasible paths that might be difficult to detect by existing techniques

However, as the presented approach heavily relies on execution data, it suffers an unsatisfiable precision when there are not adequate test cases. As the future work, the study will leverage static techniques with high precision to suppress false positives. It is also demanding to enhance our approach to deal with concurrent execution and inter-procedural infeasible paths.

REFERENCES

Aho, A.V., R. Sethi and J.D. Ullman, 1986. Compilers: Principles, Techniques and Tools. Addison-Wesley, New York, USA., ISBN-13: 9780201100884, Pages: 796.

Bertolino, A. and M. Marre, 1994. Automatic generation of path covers based on the control flow analysis of computer programs. IEEE Trans. Software Eng., 20: 885-899.

Bodik, R., R. Gupta and M.L. Soffa, 1997a. Interprocedural conditional branch elimination. Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 16-18, 1997b, Las Vegas, NV., USA., pp: 146-158.

Bodik, R., R. Gupta and M.L. Soffa, 1997b. Refining data flow information using infeasible paths. Proceedings of the 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, September 22-25, 1997, Zurich, Switzerland, pp: 361-377.

Bueno, P.M.S. and M. Jino, 2000. Identification of potentially infeasible program paths by monitoring the search for test data. Proceedings of the 15th IEEE International Conference on Automated Software Engineering, September 11-15, 2000, Grenoble, France, pp: 209-218.

Do, H., S. Elbaum and G. Rothermel, 2005. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. Empirical Software Eng., 10: 405-435.

Dufour, B., B.G. Ryder and G. Sevitsky, 2007. Blended analysis for performance understanding of framework-based applications. Proceedings of the International Symposium on Software Testing and Analysis, July 9-12, 2007, London, UK., pp: 118-128.

Goldberg, A., T.C. Wang and D. Zimmerman, 1994. Applications of feasible path analysis to program testing. Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, August 17-19, 1994, Seattle, WA., USA., pp: 80-94.

Gupta, N., A.P. Mathur and M.L. Soffa, 2000. Generating test data for branch coverage. Proceedings of the 15th IEEE International Conference on Automated Software Engineering, September 11-15, 2000, Grenoble, France, pp: 219-227.

Hall, M., E. Frank, G. Holmes, B. Pfahringer, P. Reutemann and I.H. Witten, 2009. The WEKA data mining software: An update. SIGKDD Explorations Newslett., 11: 10-18.

Harrold, M.J., B. Malloy and G. Rothermel, 1993. Efficient construction of program dependence graphs. SIGSOFT Software Eng. Notes, 18: 160-170.

Hedley, D. and M.A. Hennell, 1985. The causes and effects of infeasible paths in computer programs. Proceedings of the 8th International Conference on Software Engineering, August 28-30, 1985, London, UK., pp: 259-266.

Hovemeyer, D. and W. Pugh, 2004. Finding bugs is easy. ACM SIGPLAN Notices, 39: 92-106.

Hovemeyer, D. and W. Pugh, 2007. Finding more null pointer bugs, but not too many. Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering, June 13-17, 2007, San Diego, CA., USA., pp: 9-14.

Jasper, R., M. Brennan, K. Williamson, B. Currier and D. Zimmerman, 1994. Test data generation and feasible path analysis. Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, August 17-19, 1994, Seattle, WA., USA., pp: 95-107.

Malevris, N., 1995. A path generation method for testing lcsajs that restrains infeasible paths. Inform. Software Technol., 37: 435-441.

Mueller, F. and D.B. Whalley, 1992. Avoiding unconditional jumps by code replication. SIGPLAN Notices, 27: 322-330.

Ngo, M.N. and H.B.K. Tan, 2007. Detecting large number of infeasible paths through recognizing their patterns. Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIG-SOFT Symposium on the Foundations of Software Engineering, September 3-7, 2007, Dubrovnik, Croatia, pp: 215-224.

Ngo, M.N. and H.B.K. Tan, 2008. Heuristics-based infeasible path detection for dynamic test data generation. Inform. Software Technol., 50: 641-655.

Tan, P.N., M. Steibach and V. Kumar, 2006. Introduction to Data Mining. Pearson Addison Wesley, Boston, MA., USA., ISBN-13: 9780321420527, Pages: 769.

Visser, W., C.S. Pasareanu and S. Khurshid, 2004. Test input generation with Java PathFinder. Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, July 11-14, 2004, Boston, MA., USA., pp: 97-107.

Witten, I.H. and E. Frank, 2005. Data Mining: Practical Machine Learning Tools and Techniques. 2nd Edn., Morgan Kaufmann, New York, USA., ISBN-13: 9780120884070, Pages: 560.

Yates, D. and N. Malevris, 1989. Reducing the effects of infeasible paths in branch testing. SIGSOFT Software Eng. Notes, 14: 48-54.

Zhang, J. and X. Wang, 2001. A constraint solver and its application to path feasibility analysis. Int. J. Software Eng. Knowledge Eng., 11: 139-156.

Zhang, J., C. Xu and X. Wang, 2004. Path-oriented test data generation using symbolic execution and constraint solving techniques. Proceedings of the 2nd International Conference on Software Engineering and Formal Methods, September 28-30, 2004, Beijing, China, pp: 242-250.