# Detecting Large Number of Infeasible Paths through Recognizing their Patterns

Minh Ngoc Ngo and Hee Beng Kuan Tan
School of Electrical and Electronic Engineering
Nanyang Technological University, Singapore 639798
{ngom0002, ibktan}@ntu.edu.sg

## ABSTRACT

A great majority of program paths are found to be infeasible, which in turn make static analysis overly conservative. As static analysis plays a central part in many software engineering activities, knowledge about infeasible program paths can be used to greatly improve the performance of these activities especially structural testing and coverage analysis. In this paper, we present an empirical approach to the problem of infeasible path detection. We have discovered that many infeasible paths exhibit some common properties which are caused by four code patterns including *identical/complement-decision*, *mutually-exclusive-decision*, *check-then-do* and *looping-by-flag* pattern. Through realizing these properties from source code, many infeasible paths can be precisely detected. Binomial tests have been conducted which give strong statistical evidences to support the validity of the empirical properties. Our experimental results show that even with some limitations in the current prototype tool, the proposed approach accurately detects 82.3% of all the infeasible paths.

## Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging; D.2.4 [**Software Engineering**]: Software/Program Verification; G.4 [**Mathematical Software**]: Algorithm design and analysis;

## General Terms:

Algorithm, Experimentation, Design, Theory

## Keywords

Infeasible paths, static analysis, empirical properties, structural testing, coverage analysis

## 1. INTRODUCTION

Static analysis is undoubtedly an integral part of many software engineering activities [1, 7, 8, 15]. However, one of the biggest challenges for static analysis is the existence of infeasible program paths in that there is no input for which the paths will be executed. This is because static analysis is done before execution; one commonly made assumption during static analysis is that every program path is executable. Such conservative analysis too often yields imprecise results.

According to Hedly et al. [12] 12.5% of all the paths are infeasible. If a majority of infeasible paths can be detected during static analysis, it will greatly improve the performance of many software engineering activities, particularly structural testing and coverage analysis.

In general, detecting infeasible paths is theoretically unsolvable. Many techniques rely on symbolic evaluation [2, 8, 25], thus trading expensive computation for sharper analytical results. Moreover, due to the limitation of symbolic evaluation in handling pointers, arrays and function calls, only a small percentage of infeasible paths can be detected. Another class of approaches is based on properties of infeasible paths which are empirically discovered [3, 7, 15]. These approaches are fast and effective. However, to the best of our knowledge, none of the existing heuristics is capable of detecting a large number of infeasible paths. Dynamic test data generation algorithms can also be used to detect infeasible paths by monitoring the execution of a program [3, 10]. However, test data generation often uses symbolic execution; thus, it is almost as expensive as symbolic execution.

Although it is impossible to solve the general problem of identifying all infeasible paths, we have discovered that many infeasible paths exhibit some common properties. Motivating by this fact, we characterize in this paper the key properties of infeasible paths and four code patterns that can cause these paths including *identical/complement-decision*, *mutually-exclusive-decision*, *check-then-do* and *looping-by-flag* patterns. Based on these properties, we further proposed a static approach to detect infeasible program paths in the four code patterns. We also provide significant statistical evidences to support the validity of each property.

The remainder of the paper is organized as follows. Section 2 highlights the importance of infeasible path detection and reviews existing approaches to this problem. Section 3 presents properties of infeasible paths in four common code patterns and Section 4 statistically validates these properties. Section 5 proposes a novel approach to detect infeasible program paths. Section 6 evaluates the effectiveness of the proposed approach through experiments. Section 7 concludes the paper.

## 2. BACKGROUND

Although researchers have highlighted the significance of infeasible path detection in many software engineering fields [2, 7, 8], it is particularly useful for structural testing and coverage analysis.

Structural testing, both control-flow and data-flow testing involves three steps: (1) Identification of a set of paths that will meet the selected criterion; (2) Generating a set of test data that will cause the paths to be executed; (3) Executing the program with the test data.

The paths generated in step 1 are referred to as test requirements. Zhu et al. [27] conclude that, none of the path-based coverage criteria is applicable due to the possible existence of infeasible paths in the test requirements. Therefore, infeasible path detection can save considerable effort, both manual and automatic, that is taken to generate test data for these paths in step 2. Typically, test data generation algorithms [4, 10, 13] do not consider infeasible paths.

For path-based testing, information on infeasible path can be used to guide the algorithm which selects program paths to be tested. As such, infeasible paths can be removed from the test requirements.

In dataflow testing, if infeasible paths are not detected during static analysis, definition-use (def-use) pairs which lie on infeasible paths might be selected for testing. Weyuker [23] describes the difficulties in def-use testing as follows:

> *The problem was determining which of the definition/use associations or du-paths were executable [feasible]. This problem is encountered when using many program-based criteria, including statement and branch coverage, but is particularly acute for all-dupaths criterion since there are frequently a large number of unexecutable du-paths. In fact, we found that the unexecutable path problem, not the large number of required test cases, was the primary practical difficulty in using the all du-paths criterion.*

Coverage analysis is used to assure the quality of a test suit. Given a test coverage criterion and the test requirements with respect to the criterion, coverage analysis computes the number of paths in the test requirements that have been exercised by a test suite. This is the coverage degree of the test suite with respect to the coverage criterion. 100% coverage can rarely be achieved on real program due to the existence of infeasible paths. If the test requirements contain infeasible paths, it is not obvious what the achievable coverage degree is; thus we could not know how much more testing is needed for a particular situation.

Improving the precision of static analysis by detecting infeasible program paths has been tackled by many researchers. Clarke [5] presents an approach to test data generation based on symbolic execution. Each path is symbolically executed to generate the constraints representing the path. If a constraint is found to be inconsistent with some others then the path is shown to be infeasible.

Goldberg et al. [8] also use symbolic execution to construct a formula representing a path. The formula is solvable if and only if the path is feasible. A theorem- prover is, therefore, invokes to test the feasibility of the formula; thus the path. The approach can only be applied to a subset of Ada language. Moreover, the performance of the method is limited by the type of symbolic expressions that the theorem prover can solve.

Bodik et al. [2] propose an approach to detect infeasible paths in a program using branch correlation to improve the accuracy of traditional data flow analysis. The work is motivated by the experimental results from the authors' previous work [1], which show that 9 to 40% of conditional statements in large program exhibit correlation which can be detected statically during compile time. However, it is reported in [2] that only about 45% of the conditional statements are analyzable due to some limitations of symbolic evaluation and the compiler. Moreover, only about 13% of the analyzable conditional statements show some correlations during compile time.

Malevris [14, 15] introduces a metric to predict the feasibility of a path which is based on the observation that the greater number of predicates (conditional statements) contained within a path, the greater the probability of it to be infeasible. $\chi 2$ tests has been performed which show with strong confidence the dependency between the number of predicates and the infeasibility of a path. This approach is similar to our approach in that we also propose some empirical properties and use hypothesis testing to validate the correctness of the properties.

Forgacs et al. [7] propose using dataflow analysis and program slicing to select a set of potentially feasible paths to reach a given point in the program. However, no statistic has been given to show the effectiveness of the proposed heuristic.

Bueno et al. [3] present a tool which uses genetic algorithms to identify potentially infeasible program paths during the test data generation. However, only a small-scale experiment (40 infeasible paths) has been conducted to validate the correctness of the approach. Computational cost of the infeasible path identification is another weakness of this approach.

Before moving further, we shall review some terms that will be used throughout this paper. We shall adopt the definition of **control flow graph (CFG)** by Harrold et al. [11] to represent a program. In a CFG, a node represents a program statement and an edge represents the transfer of control between statements. Nodes are represented by circles. **Predicate nodes**, from which two edges may originate, are represented by diamonds. The **predicate** at each predicate node will be evaluated during the program execution to determine which out-coming edge of the predicate node will be traversed. There are two types of predicate nodes namely predicate nodes of selection construct (if statements) and predicate nodes of iteration construct (while statements). Each out-coming edge of a predicate node is called a **branch**. Each branch ($p$, $q$) in the CFG is labeled by a predicate, referred to as a **branch predicate**, describing the conditions under which the branch will be traversed.

In a CFG, a node $u$ **dominates** a node $w$ if and only if every path from the entry node to $w$ contains $u$. A node $w$ **post-dominates** a node $u$ if and only if every path from $u$ to the exit node contains $w$.

A node $y$ is **control dependent** on a node $x$ if and only if $x$ has successors $x'$ and $x''$ such that $y$ post-dominates $x'$ but $y$ does not post-dominates $x''$. Furthermore, we say that node $y$ is **transitively control dependent** [22] on node $x$ if there is a sequence of nodes, $x = x_0, x_1, \ldots, x_n = y$, such that $x_j$ is control dependent on $x_{j-1}$, $1 \leq j \leq n$.

Let $S$ be a set of variables. A **definition clear (def-clear) path** from node $p$ to node $q$ with respect to $S$ is a path which starts from $p$ and ends at $q$ and none of the variables in $S$ are modified along the path [13].

A **basis set** [16] of paths between nodes $p$ and $q$ contains all the path from $p$ to $q$ which are linearly independent and the paths in the basis set can be used to construct any path from $p$ to $q$. A path in this basis set is often referred to as a **basis path from *p* to *q***.

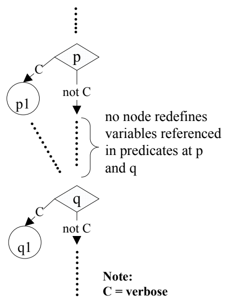
## 3. INFEASIBLE PATH PATTERNS

In this section, we characterize four code patterns and the type of infeasible paths caused by each one of them. Our experimental results show that infeasible paths detected in these code patterns constitute a very large proportion of infeasible paths.

## 3.1 Identical/complement-decision Pattern

We have discovered that, sometimes, actions to be performed under the same or completely different conditions could be implemented by separate independent selection constructs (if statements) with identical or complement conditions. This is referred to as **identical/complement-decision pattern**.

```
package soot.jimple.spark.solver;...
public final class PropWorklist
      extends Propagator {...
public final void propagate() {
......
/*p*/   if( verbose ) {
/*p1*/   G.v().out.println(...);
        }
        ...
/*q*/   if( verbose ) {
/*q1*/   G.v().out.println(...);
        }
       ...
}...
```

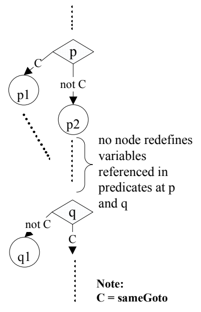


p
C
not C
p1
no node redefines variables referenced in predicates at p and q
q
C
not C
q1
Note:
C = verbose

**(a) Identical-decision pattern example** **(b) CFG**

**Figure 1. Identical-decision pattern**

```
package soot.javaToJimple.toolkits;......
public class CondTransformer
    extends BodyTransformer {...
private void transformBody(...){
 ...
/*p*/   if (sameGoto){
/*p1*/    newTarget =
            ((IfStmt)stmtSeq[5]).getTarget();
        }
        else {
/*p2*/   newTarget = next;
         oldTarget =
            ((IfStmt)stmtSeq[5]).getTarget();
        }
        ......
/*q*/   if (!sameGoto){
/*q1*/   b.getUnits().insertAfter(...);
        }
        ......
}......
```



p
C
not C
p1
p2
no node redefines variables referenced in predicates at p and q
q
not C
C
q1
Note:
C = sameGoto

**(a) Complement-decision pattern example** **(b) CFG**

**Figure 2. Complement-decision pattern**

**Definition 1 – Identical/complement-decision pattern.** Let *p* and *q* be predicate nodes of *selection construct* in a CFG. Let $p_{succ}$ and $q_{succ}$ be two successors of *p* and *q* respectively. We say that *p* and *q* follow **identical/complement-decision pattern** if and only if the following two conditions are satisfied:

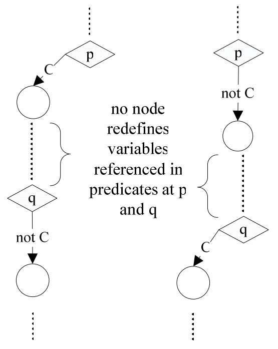
1. All the basis paths from *p* to *q* are def-clear paths with respect to variables referenced in *p* and *q*.
2. The branch predicates of ($p$, $p_{succ}$) and ($q$, $q_{succ}$) are syntactically identical.

Figure 1 and Figure 2 gives an example of identical-decision pattern and complement-decision pattern with their CFGs respectively. Both examples are taken from SOOT [20]. We use '…' to represent statements which are not important to the examples. In Figure 1(b), branch predicates of (*p*, `p1`) and (*q*, `q1`) are syntactically identical. In Figure 2(b), branch predicates of (*p*, `p2`) and (*q*, `q1`) are syntactically identical.

The implementation of this pattern leads to some infeasible paths. Figure 3 illustrates the infeasible paths that are caused by *identical/complement-decision* pattern. Next, we shall formalize the property of infeasible paths caused by *identical/complement-decision* pattern.

**Property 1 – Infeasible paths caused by identical/complement-decision pattern.** Let *p* and *q* be predicate nodes of selection construct in a CFG. If *p* and *q* follow *identical/complement-decision* pattern then any path through the CFG that contains branches of *p* and *q* with syntactically different branch predicates is infeasible.

This property is invariant and can be proved directly from the given conditions. In the first type of infeasible path (Figure 3a) the branch predicates of branches at *p* and *q* are '*C*' and 'not *C*' respectively; thus there exist no input values which can satisfy both branch predicates. Similarly, the second type of path (Figure 3b) is also infeasible.



p
C
no node redefines variables referenced in predicates at p and q
q
not C
p
not C
q
C

**(a) Path 1** **(b) Path 2**

**Figure 3. Infeasible paths in Identical/complement-decision pattern**

## 3.2 Mutually-Exclusive-Decision Pattern

Empirically, we have discovered that a decision based on the same set of factors for one purpose is often implemented in a program using two approaches.

```
package org.apache.crimson.parser;
final class ContentModel{...
public boolean first (String token) {...
 if (content instanceof String && content == token)
   retval = true;
 else if (((ContentModel)content).first (token))
   retval = true;
 else if (next != null)
   retval = ((ContentModel)next).first (token);
 else
   retval = false;
...
}...
```

**Figure 4. Nested-if pattern**

One alternative for implementing this type of decision is to use nested-if structure: the next condition will only be checked if the previous condition fails. Figure 4 illustrates this nested-if code pattern through an example taken from Crimson [6].

The second alternative avoids the use of nested-if structure. It formulates a set of mutually exclusive conditions based on the set of factors for making the decision so that a separate selection construct for each condition with only one non-null branch is used to jointly implement the decision. This design pattern is referred to as **mutually-exclusive-decision pattern**.

The implementation of this pattern leads to some infeasible paths in a program CFG. Figure 5 illustrates this design pattern through an example taken from SOOT. In this example, any path which contains more than one node from the set of nodes {`v1`, `v2`, `v3`, `v4`, `v5`} is infeasible. This is because the set of conditions at nodes `p1`, `p2`, `p3`, `p4` and `p5` are mutually exclusive; each value of `n` only satisfies at most one condition. The CFG of this code pattern is given in Figure 6(a). Figure 6(b) illustrates the type of infeasible path caused by this code pattern.

```
package soot.javaToJimple;...
public class InnerClassInfoFinder
        extends polyglot.visit.NodeVisitor{...
public polyglot.visit.NodeVisitor enter(...) {
/*p1*/  if (n instanceof polyglot.ast.LocalClassDecl) {
/*v1*/    localClassDeclList.add(n);
          }
/*p2*/  if (n instanceof polyglot.ast.New) {
/*v2*/    if (((polyglot.ast.New)n).anonType() != null){
            anonBodyList.add(n);
          }
        }
/*p3*/  if (n instanceof polyglot.ast.ProcedureDecl) {
/*v3*/    memberList.add(n);
          }
/*p4*/  if (n instanceof polyglot.ast.FieldDecl) {
/*v4*/    memberList.add(n);
          }
/*p5*/  if (n instanceof polyglot.ast.Initializer) {
/*v5*/    memberList.add(n);
          }
        return enter(n);
}...
```

**Figure 5. Mutually-exclusive-decision pattern example**

Though from code optimization and infeasible path avoidance viewpoint, the first alternative is clearly better, from our empirical observation, the mutually-exclusive decision pattern (second alternative) is frequently used. We believe this could be due to the latter alternative provides a neater code structure.

Before proceeding to the formalization of infeasible paths property in *mutually-exclusive-decision* pattern, we shall introduce some terms. A **prime variable** is a variable which is not defined through any other variable in the program. In Figure 5, `n` is a prime variable because `n` is an input, it is not defined through any other variables in the method.

Given a set of predicate nodes of selection construct $p_1,...,p_n$. The **set of external variables** of $p_j$, $1 \leq j \leq n$, contains all the variables which are defined by nodes that are control dependent on $p_j$ and are referenced by nodes that are not control dependent on $p_j$.

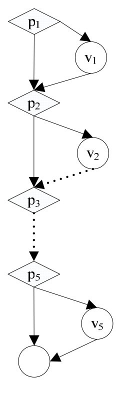
**Definition 2 – Empirical mutually-exclusive-decision pattern.** Let $p_1,..., p_n$ be different predicate nodes of selection construct in a CFG. If these predicate nodes satisfy the following conditions, then we say that $p_1,..., p_n$ are **empirically mutually exclusive** (**e-mutually-exclusive**):

1. $p_j$ dominates $p_{j+1}$ and $p_{j+1}$ post-dominates $p_j$, $1 \leq j \leq n-1$.
2. $p_j$ has only one successor that is control-dependent on $p_j$, $1 \leq j \leq n-1$.
3. Any basis path from $p_1$ to $p_j$, $1 \leq j \leq n$, is a def-clear path with respect the variables referenced at $p_j$.
4. The sets of prime variables that are referenced at $p_j$, $1 \leq j \leq n$, are identical.
5. The sets of external variables of all $p_j$, $1 \leq j \leq n$, are identical.

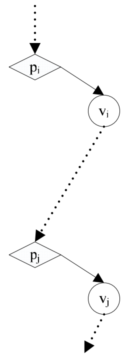
The first condition reflects that these predicate nodes define separate selections structures, which means they are not nested-if structure. The second condition ensures that each predicate node has only one non-null branch. The fourth condition reflects the fact that the decision implemented by these predicate nodes is based on the same set of factors. The last condition is to ensure that these predicate nodes are implemented for one purpose.

**Property 2 –Infeasible paths caused by mutually-exclusive decision pattern.** If $p_1$, ....., $p_n$ are e-mutually exclusive predicate nodes of selection construct in a CFG then any path that contains nodes $u$, $v$ such that $u$ and $v$ are control-dependent on $p_i$ and $p_j$ respectively ($1 \leq j, i \leq n, i \neq j$), is infeasible

The rationale behind Property 2 is the *uniqueness of decision* which means at most one decision is made at one point in time.



(a) CFG



(b) infeasible path according to Property 2

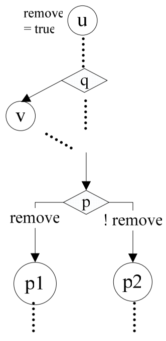
**Figure 6. Mutually-exclusive-decision pattern and its infeasible paths**

## 3.3 Check-then-do Pattern

In many situations, a set of actions is only performed upon the successful checking of a set of conditions. The commonly used design method to implement the above-mentioned requirement is to separate the checking of conditions from the actions to be performed. The checking of conditions reports the outcome through setting a 'flag' variable. Actions are performed in a branch of a predicate node which tests the value of the 'flag' variable. This design pattern is referred to as **check-then-do** pattern. Many instances of this design pattern can be found in SOOT [20].

```
package soot.dava.toolkits.base.finders;...
public class SynchronizedBlockFinder
   implements FactFinder{...
private boolean verify_CatchBody(...){...
/*u*/  boolean remove = true;
       while (pit.hasNext()) {
         ...
/*q*/      if(synchBody.contains(pAs)){
/*v*/        remove=false;
           }
       }
/*p*/  if(remove){
/*p1*/     toRemove.add(synchAs);
       }
/*p2*/ ...
}...
```

(a) Check-then-do pattern example



(b) CFG

**Figure 7. Check-then-do pattern**

*Check-then-do* pattern always introduces some infeasible paths. Figure 7 illustrates c*heck-then-do* the through an example taken from SOOT. Variable `remove` serves as a flag variable which is assigned to `true` initially. This variable is set to `false` if `synchBody.contains(pAs)` is `true`. After the checking, if value of `remove` is `true` (node `p`), node `p1` will be executed which adds `synchAs` to `toRemove`.

In this code segment, any path which contains nodes `u`, `v` and follows the true branch at node `p`, (`p, p2`), is infeasible because the predicate at node `p` is always evaluated to `false` along the path. Similarly, any path which contains node `u` but does not contain node `v` and does not follow the true branch at node `p` is also infeasible.

**Definition 3 – Empirical check-then-do pattern**. Let *K* be a variable in a program. Let *p* be a predicate node of selection construct. Let *u* be a node in the CFG. We say that *u* and *p* follow empirical **check-then-do pattern (e-check-then-do)** with respect to *K* if they satisfy the following conditions:

1. *p* only references to *K*
2. *u* dominates *p* and *p* post-dominates *u*
3. *u* assigns *K* to a constant which always results to satisfaction of the branch predicate of one branch of *p*

Next, we present an empirical property to realize the infeasible paths introduced in *check-then-do* pattern.

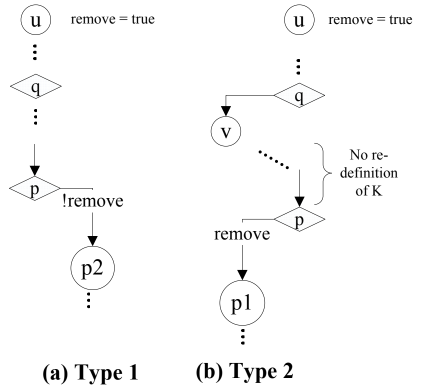


**Figure 8. Two types of infeasible paths in check-then-do pattern**

**Property 3 – Infeasible paths caused by check-then-do pattern.** Let *K* be a variable in a program. Let *p* be a predicate of selection construct and $p_{succ1}$, $p_{succ2}$ be the two successors of *p*. Let *u* be a node such that *u* and *p* follow the e-*check-then-do* pattern with respect to *K*. If the value of *K* assigned at *u* always results to the satisfaction of ($p$, $p_{succ2}$), then the following paths are infeasible:

1) Any path that contains *u*, *p*, $p_{succ2}$ and the sub-path from *u* to *p* contains a node *v* that redefines *K* to a value syntactically different from the value assigned by *u* and no redefinition of *K* after this node in the sub-path.
2) Any path that contains *u*, *p*, $p_{succ1}$ and the sub-path from *u* to *p* does not contain any node that defines/refines *K* to a value syntactically different from the value assigned by *u*.

The rationale of Property 3 is as follows. The given condition implies that the value of *K* defined/redefined at *u* always results to the satisfaction of the branch predicate of one branch of *p*. As *v* redefines *K* to a value syntactically different from the value assigned at *u* it is likely that the values of *K* defined/redefined at *v* always results to the satisfaction of the branch predicate of the other branch of *p*. Therefore, *K* serves as a flag variable to report the result of checking with one reports "success" and the other reports "fail". Then, the selection construct defined by *p* checks the value of *K* to determine what the actions to be performed.

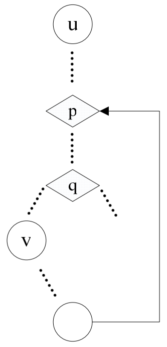
Figure 8 illustrates the two types of infeasible paths introduced by *check-then-do* pattern.

## 3.4 Looping-by-Flag Pattern

Sometimes, programmers use a 'flag' variable to control the termination of a loop. Normally, the flag variable is initialized to a value which enables the execution of the loop. Inside the body of the loop, the flag variable will be set to another value if some conditions are satisfied. This value of the flag variable will lead to the termination of the loop. This coding pattern is often referred to as **looping-by-flag pattern**. Figure 9(a) describes the pattern through an example from SOOT and Figure 9(b) gives the corresponding CFG. In Figure 9, variable `change` serves as a flag variable to control the loop at node `p`.

```
package soot.javaToJimple.toolkits;...
public class CondTransformer
       extends BodyTransformer {...
protected void internalTransform(...){...
/*u*/   boolean change = true;
/*p*/   while (change){
          ...// no redefinition of change
/*q*/     while (it.hasNext()){
/*v*/       change = false;
             Stmt s = (Stmt)it.next();
             if (testStmtSeq(s, pos)) {
              pos++;
             }
             if (pos == 6) {
              change = true;
              break;
             }
          }...
        }...
}...
```

**(a) Loopping-by-flag pattern example**



**(b) CFG**

**Figure 9. Looping-by-flag pattern**

Below, we formalize *looping-by-flag* pattern.

**Definition 4 – Empirical looping-by-flag pattern**. Let *K* be a variable in a program. Let *u*, *v*, *p* and *q* be four nodes in the CFG of the program in which *p* is a predicate node of iteration construct and *q* is a predicate node of selection construct. We say that *u*, *v*, *p*, *q* follow **empirical looping-by-flag pattern (e-looping-by-flag)** with respect to *K* if they satisfy the following properties:

1) The predicate at *p* only references *K*.
2) *q* is transitively control-dependent on *p*.
3) *u* dominates *p* and *p* post-dominates *u*.
4) *u* assigns *K* to a constant $k_0$, which always results to satisfaction of the branch predicate of the entry branch of *p*.
5) *v* is control dependent on *q* and *v* assigns *K* to a value that is syntactically different from $k_0$.

**Property 4 – Infeasible paths caused by looping-by-flag pattern.** Let *K* be a variable in a program. Let *u*, *v*, *p* and *q* be four nodes in the CFG of the program in which *p* is a predicate node of iteration construct and *q* is a predicate node of selection construct. Let ($p$, $p_{succ}$) be the entry branch at *p*. If *u*, *v*, *p*, *q* follow *e-looping-flag* pattern with respect to *K*, then any path π that satisfies the following conditions is infeasible:

*1)* π contains *u*, *v* and *q*
2) π contains more than one appearance of $p_{succ}$
*3)* At least one sub-path from *v* to an appearance of $p_{succ}$ is a def-clear path with respect to *K*

The rationale behind property 4 is that if *v* assigned *K* to a value syntactically different from *u* it is likely that the value will lead to the termination of the loop controlled by *p*. As such, a path in which there is no redefinition of *K* along a sub-path from *v* to an appearance of $p_{succ}$ is likely to be infeasible.

According to Property 4, the following paths in Figure 9(a) are infeasible : $\{ (..., u, p, p_{succ} ..., q, v, \underbrace{............, p, p_{succ}}_{\text{no redefinition of K}}, ....) \}$.

## 4. STATISTICAL VALIDATION

In this section, we report the results of our binomial testing [9] to statistically validate empirical properties presented in the previous section namely Property 2, 3 and 4. As Property 1 is an invariant property, it is excluded from our statistical validation. For each empirical property $\Re$, the alternate hypothesis states that the property holds for equal or more than 97% of the cases. And, therefore, the null hypothesis states that the property holds for less than 97% of the cases. That is:

- $H_0$ (null hypothesis): $p(\Re \text{ holds}) < 0.97$
- $H_1$ (alternate hypothesis): $p(\Re \text{ holds}) \geq 0.97$

The binomial test statistics $z$ is computed as $z = \frac{X/n - p}{\sqrt{(p(1-p)/n)}}$, where $n$ is the total sample size for the test and $X$ is the number of cases that support the alternative hypothesis $H_1$. Taking 0.001 as the type I error, if $z > 3.1$, we reject the null hypothesis, otherwise, we accept the hypothesis.

**Table 1. Samples for testing empirical properties**

| System | KLOC | Source | $P_2$ | $P_3$ | $P_4$ |
|---|---|---|---|---|---|
| Systems written in Java | | | | | |
| AVIEC | 31.2 | Student | 67 | 51 | 21 |
| Taxier | 67.5 | Student | 98 | 74 | 35 |
| HtmlParser | 61.6 | Open source | 121 | 134 | 87 |
| Jhotdraw | 71.7 | Open source | 34 | 51 | 47 |
| JlibSys | 12.2 | Open source | 32 | 19 | 36 |
| NCS | 147.4 | Industrial | 148 | 174 | 81 |
| System written in PHP | | | | | |
| NetOfiice | 42.6 | Open source | 59 | 42 | 32 |
| OpenIT | 37.8 | Open source | 43 | 23 | 14 |
| Conferencer | 14.9 | Student | 27 | 19 | 23 |
| Total | | | 629 | 587 | 376 |

Our samples were randomly drawn from system spread across a wide variety of application domains and are written by programmers of different levels. Each procedure forms a case (an individual in the sample) for the testing of Property 2, 3 and 4.

Table 1 gives the sample size for the testing of each property. The first two columns give the name of each system and the source from where the system is taken respectively. Note that all the open source systems were downloaded from [21] by searching for the system's name. NCS is a J2EE application developed by the National Computer System of Singapore. For confidential reason, we do not disclose the system's name here. Columns $P_2$, $P_3$ and $P_4$ give the number of cases drawn from each system for the testing of Property 2, 3 and 4 respectively.

All the cases in the sample for testing each hypothesis gave affirmative result. Therefore, the z-score of binomial test for Property 2, 3 and 4 are 4.41 (X = 629, $n$ = 629, $p$ = 0.97), 4.26 (X = 587, $n$ = 587, $p$ =0.97) and 3.41 (X = 376, $n$ = 376, $p$ =0.97) respectively. As all the z-cores are greater than 3.1, we conclude that Property 2, 3 and 4 hold for equal or more than 97% of all the cases at 0.1% level of significance.

## 5. DETECTING INFEASIBLE PATHS

Based on the properties established in Section 3, we present in this section a set of algorithms to detect infeasible program paths.

Algorithm `detect_infeasible_paths` (Figure 10) is the main algorithm which detects infeasible paths in a set Π of paths in a program *P*. The algorithm sequentially calls `detect_I/CD_pattern`, `detect_e-MED_pattern`, `detect_e-CTD_pattern` and `detect_e-LBF_pattern` algorithms to detect infeasible paths introduced by *identical/complement-decision* pattern, *mutually-exclusive-decision* pattern, *check-then-do* pattern and *looping-by-flag* pattern respectively.

```
Algorithm detect_infeasible_paths(program P, set of path Π)
Output: a set of infeasible paths in Π.
Begin
1. Construct control flow graph G of P
2. $I_1$ = detect_I/CD_pattern(G, Π)
3. $I_2$ = detect _e-MED_pattern(G, Π)
4. $I_3$ = detect _e-CTD_pattern(G, Π)
5. $I_4$ = detect _e-LBF_pattern(G, Π)
6. return ($I_1 \cup I_2 \cup I_3 \cup I_4$)
End
```

**Figure 10. Detect infeasible paths in a given set of paths**

Each of the algorithms in Figure 11, Figure **12**, Figure 13 and Figure 14 consists of two steps. In the first step, the pattern is realized based on the definition using purely static analysis. In the second step, infeasible paths in the given set of paths are detected by using the properties of infeasible paths caused by the corresponding pattern. Below we describe the main features of each algorithm.

```
Algorithm detect_I/CD_pattern(G, Π)
Output: a set of infeasible paths in Π detected by identical-decision pattern.
Begin
// Step 1
1. Pre = {p | p is a predicate node of selection construct}
2. $\Im_{ID} = \varnothing$, $\Pi_{infeasible} = \varnothing$
3. for (each pair (p, q), $p \in$ Pre, $q \in$ Pre)
4.     if (all the basis path from p to q are definition-clear
       path w.r.t variables referenced at p, q) then
5.         if (branch predicates of ($p$, $p_{true}$) ,($q$, $q_{true}$) are syntactically
           identical) then
6.             $\Im_{ID} = \Im_{ID} \cup \{(p_{true}, q_{false}), (p_{false}, q_{true})\}$
7.         else if (branch predicates of ($p$, $p_{true}$), ($q$, $q_{false}$) are
       syntactically identical) then
8.             $\Im_{ID} = \Im_{ID} \cup \{(p_{true}, q_{true}), (p_{false}, q_{false})\}$
       endIf
   endFor
// Step 2
9. for (each path π in Π) do
10.    for (each pair $(p, q) \in \Im_{ID}$) do
11.        if (π contains both p and q) then
12.            $\Pi_{infeasible} = \Pi_{infeasible} \cup \pi$
13. return $\Pi_{infeasible}$
End
```

**Figure 11. Detect infeasible paths in identical/complement-decision pattern**

The basic idea of algorithm `detect_I/CD_pattern` (Figure 11) is to form in the first step the set $\Im_{ID}$ of pairs of nodes ($p_{succ}$, $q_{succ}$) such that if a path contains both $p_{succ}$ and $q_{succ}$, it is infeasible. Once this has been done, infeasible paths can be easily detected in the second step.

This algorithm bases mainly on Property 1. For each pair ($p$, $q$) of predicate nodes that satisfy the first condition of *identical/complement-decision* pattern, the algorithm determines whether $p$ and $q$ follow identical-decision or complement-decision pattern. If branch predicates of the true branch of $p$, $(p, p_{true})$, and the true branch of $q$, $(q, q_{true})$, are identical, $p$ and $q$ actually follow *identical-decision* pattern (line 5). According to property 1, any path which contains $(p, p_{true})$ and $(q, q_{false})$ and any path which contains $(p, p_{false})$ and $(q, q_{true})$ are infeasible. As such, $(p_{true}, q_{false})$ and $(p_{false}, q_{true})$ are added to $\Im_{ID}$ (line 6). If $p$ and $q$ do not follow *identical-decision* pattern, similar checking is done for *complement-decision* pattern (line 8, 9).

```
Algorithm detect_e-MED_pattern(G, Π)
Output: a set of infeasible path in Π detected by mutually-exclusive-decision pattern.
Begin
// Step 1
1.  Pre = {$p_j$ | 1≤ j ≤ n, $p_j$ is a predicate node of selection construct, pi
    has only one successor which is control dependent on $p_j$, $p_{j-1}$
    dominates $p_j$, $p_j$ post-dominates $p_{j-1}$}, $\Pi_{infeasible} = \emptyset$
2.  for (each predicate $p_i \in Pre$, 1≤ i ≤ n) do
3.      if (all the basis paths from $p_1$ to $p_i$ are def-clear path w.r.t
            variables referenced at $p_i$) then
4           remove $p_i$ from Pre and continue,
5.      $K_{p_i}$ = {prime variables referenced at $p_i$}
6.      $V_{p_i}$ = {external variables of $p_i$}
    endFor
7.  $Partition = \{B \mid B \subseteq Pre, p \in B$ and $q \in B$ iff $K_p = K_q$ & $V_p = V_q\}$
// Step 2
8.  for (each path π in Π) do
9.      for (each $B \in Partition$ which |B| > 1) do
10.         $\Im_{MED}$= {$p_{succ}$ | $p_{succ}$ is control dependent on $p$, $p \in B$}
11.         if not (π contains more than one element in $\Im_{MED}$) then
12.             $\Pi_{infeasible} = \Pi_{infeasible} \cup \pi$
        endFor
    endFor
13. return $\Pi_{infeasible}$
End
```

**Figure 12. Detect infeasible paths in mutually-exclusive-decision pattern**

Algorithm `detect_e-MED_pattern` (Figure 12) first forms the set *Pre* of predicate nodes of selection construct which satisfy the first three conditions of e-*mutually-exclusive-decision* pattern (line 1→6). The objective is that *Pre* will then be partitioned into a set of subsets such that all the predicate nodes in each subset satisfy the last two conditions of *e-mutually-exclusive-decision* pattern (line 7). Therefore, each subset in the partition contains a set of e-mutually-exclusive predicate nodes. This is the most important step of this algorithm. For each set of e-mutually-exclusive predicate node, Step 2 just applies Property 2 to each path in Π to determine its feasibility.

Algorithm `detect_e-CTD_pattern` (Figure 13) first constructs the set $\Im_{CTD}$ of all tuples $(K, u, p, p_{succ})$ such that $u$, $p$ follow the empirical *check-then-do* pattern with respect to $K$ and the value of $K$ defined by $u$ always lead to the execution of the branch $(p, p_{succ})$ (line 1→7). This set is used in Step 2 to detect infeasible paths as any path which satisfies Property 3 with respect a tuple in $\Im_{CTD}$ is infeasible. Basically the algorithm forms set $\Im_{CTD}$ as follow. For each predicate node $p$, line 5 performs constant substitution for predicate $C$ at $p$ by using $k_0$, which is the value assigned to $K$ at node $u$. If $C$ is evaluated to true, which means the value $k_0$ assigned to $K$ at $u$ always lead to the satisfaction of the branch $(p, p_{true})$; thus the tuple $(K, u, p, p_{true})$ is added to $\Im_{CTD}$ (line 6). Otherwise the tuple $(K, u, p, p_{false})$, where $(p, p_{false})$ is the false branch of $p$, is added to $\Im_{CTD}$ (line 7).

```
Algorithm detect_e-CTD_pattern(G, Π)
Begin
// Step 1
1.  Pre = {p | p is a predicate node of selection construct, p references
    only a single variable}, $\Im_{CTD} = \emptyset$, $\Pi_{infeasible} = \emptyset$
2.  for (each predicate p in Pre) do
3.      C = the predicate at p, K = the variable referenced by p
4.      if (there exists a node u such that u defines K to a constant $k_0$
        and u dominates p and p post-dominates u) then
5.          value = evaluate C using $k_0$
6.          if (value) then $\Im_{CTD} = \Im_{CTD} \cup \{(K, u, p, p_{true})\}$
7.          else $\Im_{CTD} = \Im_{CTD} \cup \{(K, u, p, p_{false})\}$
        endIf
    endFor
// Step 2
8.  for (each path π in Π) do
9.      for (each tuple $(u, p, p_{succ}, K) \in \Im_{CTD}$) do
10.         if (π contains $u, p, p_{succ}$ and the sub-path from u to p is a
            def-clear path w.r.t K) then $\Pi_{infeasible} = \Pi_{infeasible} \cup \pi$
11.         if (π contains $u, p, p_{succ'}$ and the sub-path from u to p contains
                a node v which redefines K and the sub-path from v to p is
                a def-clear path w.r.t K) then $\Pi_{infeasible} = \Pi_{infeasible} \cup \pi$
        endFor
    endFor
12. return Π
End
```

**Figure 13. Detect infeasible paths in check-then-do pattern**

```
Procedure detect_e-LBF_pattern(G, Π)
Begin
// Step 1
1.  Pre = {p | p is a predicate node of iteration construct, p references
    only a single variable}, $\Im_{LBF} = \emptyset$ , $\Pi_{infeasible} = \emptyset$
2.  for (each predicate p in Pre) then
3.      C = the predicate at p, K = the variable referenced by p
4.      if (there exists a node u such that u defines K to a constant k0
            and u dominates p and p post-dominates u) then
5.          if (there exist a node v which define K to a value
                syntactically different from $k_0$) then
6.              q = a node on which v is control-dependent
7.              if (q is transitively control-dependent on p) then
8.                  value = evaluate C using $k_0$
9.                  if (value) then $\Im_{LBF} = \Im_{LBF} \cup (u, v, p, q, K)$
                endIf
            endIf
        endIf
    endFor
// Step 2
10. for (each path π in Π) do
11.     for (each tuple (u, v, p, q, K) in $\Im_{LBF}$) do
12.         if (π contains u, v, q and at least two appearances of p s.t
        the sub-path from v to one appearance of p is a def-clear
        path w.r.t K)then $\Pi_{infeasible} = \Pi_{infeasible} \cup \pi$
13. return $\Pi_{infeasible}$
End
```

**Figure 14. Detect infeasible paths in looping-by-flag pattern**

Algorithm `detect_e-LBF_pattern` (Figure 14) constructs in the first step the set $\Im_{LBF}$ of tuples $(u, v, p, q, K)$ such that $u$, $v$, $p$ and $q$ follow *e-looping-by-flag* pattern with respect to $K$ and the value $k_0$ assigned to $K$ at $u$ always leads to the execution of the entry branch of the predicate node $p$. In the second step, for each tuple ($u$, $v$, $p$, $q$, $K$) in $\Im_{LBF}$, the algorithm checks whether there is any path $\pi$ which satisfies three conditions of Property 4 with respect to $u$, $v$, $q$, $p$ and $K$ (line 12). If so, $\pi$ is infeasible; thus $\pi$ is added to the set of infeasible paths detected $\Pi_{infeasible}$.

**Comparison**

Our approach consists of two steps, both of them are based purely on static program analysis techniques specifically control and data flow analysis. Only constant substitution is needed for algorithms `detect_e-CTD_pattern` and `detect_e-LBF_pattern`. Empirical properties of infeasible paths are the key features of our approach.

Many approaches rely on symbolic evaluation [2, 8, 26]. To check whether a path is infeasible, the path is symbolically executed to generate a symbolic expression representing the path. This expression is then solved by a constraint solver to determine the infeasibility of the path. Symbolic execution is expensive in both speed and space. Without dwelling on the limitations of symbolic evaluation in handling arrays, loops, pointers and function calls, not all symbolic expressions are solvable. In opposite to symbolic approach, we do not need to symbolically execute the path. Only simple constant substitution is needed. Basically, our approach checks whether the path falls into any type of infeasible paths caused by the four code patterns. As the proposed property of infeasible paths (Property 1$\rightarrow$4) relies mainly on control and data dependency information, the checking can easily be done by going through all nodes in the path to make sure that they follow certain control and data dependency relationships.

The performance of our approach is comparable to those approaches which are based on heuristics that have been empirically validated [7, 24]. However, we improve on the earlier approaches in that we provide four empirical properties (heuristics) which cover a large portion of infeasible paths.

**Time complexity**

The cost of our technique can be divided between the code pattern recognition (Step 1) and infeasible paths detection in each code pattern (Step 2). Algorithm `detect_I/CD_pattern` takes O($N^2B$) computation steps to find the set $\Im_{ID}$, where $N$ is the number of predicate nodes of selection construct in the CFG and $B$ is the number of basis paths through the CFG. According to the McCabe [16] complexity number, $B$ is equal to ($E - V + 2$) where $V$ is the number of nodes and $E$ is the number of edges of the CFG. Therefore, the complexity of the first step is O($N^2E$). In Step 2 of the algorithm, all the infeasible paths in a given set $\Pi$ can be detected in O($MP$), where $M$ is the number of paths in $\Pi$ and $P$ is the number of pairs in $\Im_{ID}$. The number of elements in $\Im_{ID}$ is generally less than $N^2$; As such, the overall complexity of algorithm `detect_ID_pattern` is O($N^2$*max($E$, $M$)).

Similarly, the cost of the first step and the second step of algorithm `detect_e-MED_pattern` is O($NE$) and O($NM$) respectively. The overall complexity of the algorithm is O($N$*max($E$,$M$)). The cost of finding infeasible paths in algorithm `detect_e-CTD_pattern` is O($NV$) and O($NM$) for the first step and the second step respectively; thus the overall complexity is O($N$*max($V$,$M$)). Algorithm `detect_e-LBF_pattern` takes O($NV^2$) for the first step and O($NM$) for the second step. Overall, the algorithm detects all infeasible paths in *loop-by-flag* pattern in O($NV^2$) computation steps.

## 6. EVALUATION

### 6.1 Experiment

To evaluate the effectiveness of the proposed approach we have applied the approach to detect infeasible paths in many programs written in Java. To ensure the generality of the proposed approach, we choose systems from various application domains. Moreover, they are written by programmers of different levels from undergraduate students to researchers to industrial software engineers. Table 2 lists all the systems and the packages in each system used for our experiment. SOOT [20] is a java byte code optimization tool. DFNET and GraphAlgo are student projects; one is a graphical dataflow analysis tool and the other is a tool to aid the teaching of data structures and algorithms. InsectJ [19] is a java byte code instrumentation framework. JTrade [21] and Crimson [6] are both open source system. JTrade is trade list management system while Crimson is a Java XML parser. JGEditor is another system provided by the National Computer System of Singapore.

**Table 2. Descriptions of target systems**

| System | KLOC | Source | Packages |
|---|---|---|---|
| SOOT | 220 | Research Prototype | soot.coffi ;<br>soot.dava.internal.AST<br>soot.dava.toolkits.base*<br>soot.javaToJimple.*<br>soot.jimple<br>soot.jimple.parser<br>soot.jimple.spark.solver<br>soot.jimple.toolkits.* |
| DFNET | 48.5 | PhD Project | ntu.dfnet<br>ntu.dfnet.graph |
| InsectJ | 40 | Research Prototype | edu.gatech.cc.rtinsect.* |
| JTrade | 8 | Open source | sfljtse<br>sfljtse.quotes<br>sfljtse.stats |
| GraphAlgo | 6 | Msc Project | add; cmst; drop |
| Crimson | 30 | Open source | org.apache.crimson.* |
| JGEditor | 122 | Industrial | ncs<br>ncs.graph<br>ncs.plaf.basic |

To facilitate the experiments, we developed a prototype tool by extending SOOT [18], a powerful open-source tool to analyze Java byte code. SOOT provides basic program analysis functions such as control flow analysis; dominance/post-dominance analysis. We implemented all the algorithms for detecting infeasible paths presented in Section 5. In algorithms `detect_e-CTD_pattern` and `detect_e-LBF_pattern`, we need to perform constant substitution for predicates which reference to only a single variable (line 5 in Figure 13 and line 8 in Figure 14). Currently, only simple constant substitution algorithms are implemented in the tool including evaluation for arithmetic operations and bitwise operations. However, we emphasize that our infeasible path detection technique supports the analysis of arbitrary predicates. In addition, we also implemented an algorithm to find the set of basis paths in a control flow graph [17].

For each system, we randomly picked some packages for the experiment. After that, we carefully chose in each package only methods which contain infeasible paths. For each such method, we computed the basis set of its CFG. We then invoked the proposed algorithms to detect all the infeasible paths in the basis set. For each basis path π, we examined the corresponding source code to determine the feasibility of the path. If π was feasible and the prototype tool concluded that it is infeasible, π was counted as a *false-positive* case. If π is infeasible and the prototype tool could not detect it, π was counted as a *true-negative* cases.

## 6.2 Results

Table 2 shows the results of our experiments on seven systems. The first column gives the name of each system. Columns $\sum$, $\sum_{truly}$, $\sum_{detected}$ give the total number of basis paths taken into consideration, the number of truly infeasible basis paths and the number of infeasible basis paths detected by the prototype tool respectively. Columns $\sum_{FP}$ and $\sum_{TN}$ give the number of false positive cases and the number of true negative cases respectively. Finally, columns $\sum_1$, $\sum_2$, $\sum_3$ and $\sum_4$ give the number of infeasible basis paths detected by Property 1, Property 2, Property 3 and Property 4 respectively.

**Table 3. Experimental results**

| System | $\sum$ | $\sum_{truely}$ | $\sum_{detected}$ | $\sum_{FP}$ | $\sum_{TN}$ | $\sum_1$ | $\sum_2$ | $\sum_3$ | $\sum_4$ |
|---|---|---|---|---|---|---|---|---|---|
| SOOT | 3240 | 1161 | 989 | 0 | 172 | 326 | 139 | 509 | 15 |
| DFNET | 438 | 222 | 198 | 0 | 24 | 132 | 32 | 30 | 4 |
| InsectJ | 275 | 76 | 76 | 0 | 0 | 12 | 14 | 50 | 0 |
| Jtrade | 216 | 112 | 77 | 0 | 35 | 65 | 0 | 12 | 0 |
| GraphAlgo | 405 | 196 | 146 | 0 | 50 | 0 | 41 | 102 | 3 |
| Crimson | 918 | 330 | 223 | 0 | 107 | 51 | 62 | 110 | 0 |
| JGEditor | 471 | 179 | 164 | 0 | 15 | 36 | 105 | 23 | 0 |
| **TOTAL** | **5963** | **2276** | **1873** | **0** | **403** | **622** | **393** | **836** | **22** |

According to Table 3, the prototype tool detected 1873 infeasible paths. There was no false positive case. As such the proposed approach successfully detected 82.3% of all the infeasible paths. The breakdown of percentage of infeasible paths detected by each property over all the infeasible paths detected is as follows: 33.2% were detected by Property 1, 21% were detected by Property 2, 44.6% were detected by Property 3 and only 1.2% was detected by Property 5.

Also from Table 2, there were 403 true negative cases, which accounts for 17.7% of all the truly infeasible basis paths. We investigated all the true negative cases and found that 95 cases, which accounts for 4.2% of all the truly infeasible paths, were not detected by the proposed approach at all. By "not detected by the proposed approach at all" we mean that those cases do not follow any of the four proposed code patterns. These code patterns are very rare. Some of them are highlighted in Figure 15 and Figure 16. Currently, we are working on explaining the intention of the programmers in these code patterns. We are also conducting more experiments to search for the instances of these code patterns.

The code pattern in Figure 15 leads to some infeasible paths. For example, line 3 and line 12 assign `sRes` and actual to the same value, which is `"Public"`. Consequently, predicate `!sRes.equals(actual)` at line 19 is always evaluated to false along any path which contains lines 3, 12 and 19; thus any path which contains lines 3, 12 and follow the true branch at line 19, (19, 20), is infeasible.

At a glance, predicates at lines 1, 3, 7 in Figure 16 look similar to *e-mutually-exclusive-decision* pattern. However, since the set of prime variables referenced in lines 1, 3, 7 are {`dest`, `this`, `other`}, {`des`, `this`}, {`dest`, `other`} respectively, these predicate does not follow *e-mutually-exclusive-decision* pattern. However, this code pattern also leads to some infeasible paths. Line 2 is only executed only if both predicates `dest != this` and `dest != other` are evaluated to true. Therefore, if line 2 is executed, the predicates at lines 3 and 7 are also true. As such, any path which contains line 2 but does not contain line 4 or/and line 8 is infeasible.

```
package soot.jimple.toolkits.annotation.qualifiers;...
public class TightestQualifiersTagger
       extends SceneTransformer {...
private void handleFields(){...
1.   String sRes = "Public";
2.   if (result == RESULT_PUBLIC){
3.    sRes = "Public";}
4.   else if (result == RESULT_PROTECTED){
5.    sRes = "Protected";}
6.   else if (result == RESULT_PACKAGE){
7.    sRes = "Package";}
8.   else if (result == RESULT_PRIVATE){
9.    sRes = "Private";}

10.  String actual = null;
11.  if (Modifier.isPublic(f.getModifiers())){
12.   actual = "Public";}
13.  else if (Modifier.isProtected(f.getModifiers())){
14.   actual = "Protected";}
15.  else if (Modifier.isPrivate(f.getModifiers())){
16.   actual = "Private";}
17.  else {
18.   actual = "Package";}

19.  if (!sRes.equals(actual)){
20.   ...
     }...
}...
```

**Figure 15. A case not detected by the proposed approach**

```
package soot.toolkits.scalar;...
public abstract class AbstractFlowSet
        implements FlowSet {...
public void union(FlowSet other, FlowSet dest) {
1.  if (dest != this && dest != other)
2.    dest.clear();

3.  if (dest != this) {
4.    Iterator thisIt = toList().iterator();
5.    while (thisIt.hasNext())
6.      dest.add(thisIt.next());
    }

7.  if (dest != other) {
8.    Iterator otherIt = other.toList().iterator();
9.    while (otherIt.hasNext())
10.     dest.add(otherIt.next());
    }
  }...
```

**Figure 16. A case not detected by the proposed approach**

The rest 308 true negative cases were not detected because of the limitations of the current prototype tool. In the current tool, for algorithms `detect_e-CTD_pattern` and `detect_e-LBF_pattern`, we only implemented simple constant substitution for predicates with arithmetic and bitwise operations. As such, code segment like the one in Figure 17 cannot be detected.

```
package soot.dava.toolkits.base.AST.transformations;...
public class ForLoopCreationHelper{...
private List createNewStmtSeqNodeAndGetInit(List commonVars){
 List stmts = new ArrayList();     // node u
 ...
 int stmtNum=0;
 while(stmtNum<currentLowestPosition   && stmtIt.hasNext()){
     stmts.add(stmtIt.next());     // node v
     stmtNum++;
 }
 if(stmts.size()>0){               // node p
     newStmtSeqNode = new ASTStatementSequenceNode(stmts);
 }
 else{
     newStmtSeqNode = null;
 }...
}...
```

**Figure 17. A case not detected by the prototype tool**

Nodes `u`, `v` and *p* in the code segment actually follow *check-then-do* pattern with respect to `stmts.size`. Indeed, when `stmts` is initialized to a new `ArrayList` at node u, variable `stmts.size` is assigned to zero. When a new element is added to `change` at node v, `stmts.size` increases by one. Later on, node *p* checks `stmts.size` against zero. However, in this case, the tool does not possess such knowledge. In addition, the tool was not able to evaluate the predicate at *p*; as such, it failed to detect infeasible paths in the code segment. Currently, we are working on extending the tool to improve these weaknesses. Knowledge about predefined functions can be stored in a knowledge base and updated by users.

## 7. CONCLUSIONS

In this paper, we have proposed a novel approach to identify infeasible paths in four common code patterns through realizing some common properties of these paths from source code. Our binomial testing shows that all the proposed properties of infeasible paths hold for 97% of all the cases at 0.1% level of significance. We have also conducted experiments to evaluate the effectiveness of the proposed approach. Even with some limitations in the current prototype tool, the proposed approach accurately detected 82.3% of all the infeasible paths in the set of basis paths in seven systems.

The novelty of our approach lies in the use of empirical properties, which provide a simple yet efficient approach to identify infeasible program paths. The use of empirical properties that have been statistically validated is used very often in medicine. However, it is not very common in software engineering. We believe that this is a promising future direction, which opens a new avenue for improving the efficiency of many software engineering activities, especially testing. We will also perform experiments at a larger scale to validate the proposed empirical properties so that it can be used with high confidence to detect most of the infeasible program paths.

## REFERENCES

[1] R. Bodik, R. Gupta, and M. L. Soffa, "Interprocedural conditional branch elimination," PLDI, Jun 15-18 1997, Las Vegas, NV, USA, 1997.

[2] R. Bodik, R. Gupta, and M. L. Soffa, "Refining data flow information using infeasible paths," *ESEC/FSE '97. 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, 22-25 Sept. 1997*, vol. 22, pp. 361-77, 1997.

[3] P. M. S. Bueno and M. Jino, "Identification of potentially infeasible program paths by monitoring the search for test data," ASE, 11-15 Sept. 2000, Grenoble, France, 2000.

[4] C. Cadar and D. Engler, "Execution Generated Test Cases: How to Make Systems Code Crash Itself," *ACM Symposium on Operating System Principle*, 2005.

[5] L. A. Clarke, "A system to generate test data and symbolically execute programs," *IEEE Transactions on Software Engineering*, vol. SE-2, pp. 215-22, 1976.

[6] Crimson, "A java XML parser," *http://xml.apache.org/crimson/.*

[7] I. Forgacs and A. Bertolino, "Feasible test path selection by principal slicing," presented at ESEC/FSE '97, Zurich, Switzerland, 1997.

[8] A. Goldberg, T. C. Wang, and D. Zimmerman, "Applications of feasible path analysis to program testing," (ISSTA), 17-19 Aug, Seattle, WA, USA, 1994.

[9] F. J. Gravetter and L. B. Wallnau, *Essentials of statistics for the behavioral sciences*. Wadsworth, Belmont, CA, 2000.

[10] N. Gupta, A. P. Mathur, and M. L. Soffa, "Generating test data for branch coverage," presented at Proceedings of ASE 2000 15th IEEE International Automated Software Engineering Conference, 11-15 Sept. 2000, Grenoble, France, 2000.

[11] M. J. Harrold, B. Malloy, and G. Rothermel, "Efficient construction of program dependence graphs," *International Symposium on Software Testing and Analysis (ISSTA), 28-30 June 1993*, vol. 18, pp. 160-70, 1993.

[12] D. Hedley and M. A. Hennell, "The causes and effects of infeasible paths in computer programs," 8th International Conference on Software Engineering (Cat. No.85CH2139-4), 28-30 Aug. 1985, London, UK, 1985.

[13] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, vol. 16, 1990.

[14] N. Malevris, "A path generation method for testing LCSAJs that restrains infeasible paths," *Information and Software Technology*, vol. 37, pp. 435-41, 1995.

[15] N. Malevris, D. F. Yates, and A. Veevers, "Predictive metric for likely feasibility of program paths," *Information and Software Technology*, vol. 32, pp. 115-18, 1990.

[16] T. J. McCabe, "Structured testing: A software testing methodology using the cyclomatic complexity metric," Nat. Bur. Stand., Washington, DC, USA 1982/12/ 1982.

[17] J. Poole, "Method to Determine a Basis Set of Paths to Perform Program Testing.," United States 1995/11/ 1995.

[18] R. V. Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot - a Java bytecode optimization framework," in *Proceedings of the 1999 conference of the Centre for Advanced Studies on Collaborative research*. Mississauga, Ontario, Canada: IBM Press, 1999, pp. 13.

[19] A. Seesing and A. Orso, "InsECTJ: a generic instrumentation framework for collecting dynamic information within Eclipse " in *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange* San Diego, California ACM Press, 2005 pp. 45-49

[20] SOOT, "A Java bytecode optimization framework," *http://www.sable.mcgill.ca/soot/.*

[21] Sourceforge, "Open-source website," *http://sourceforge.net/.*

[22] M. Weiss, "Transitive closure of control dependence: The iterated join," *ACM Letters on Programming Languages and Systems*, vol. 1, pp. 178-190, 1992.

[23] E. J. Weyuker, "The cost of data flow testing: an empirical study," *IEEE Transactions on Software Engineering*, vol. 16, pp. 121-8, 1990.

[24] D. F. Yates and N. Malevris, "Reducing the effects of infeasible paths in branch testing," presented at ACM SIGSOFT '89, Key West, FL, USA, 1989.

[25] M. Young and R. N. Taylor, "Combining static concurrency analysis with symbolic execution," *IEEE Transactions on Software Engineering*, vol. 14, pp. 1499-511, 1988.

[26] J. Zhang and X. Wang, "A constraint solver and its application to path feasibility analysis," *International Journal of Software Engineering and Knowledge Engineering*, vol. 11, pp. 139-56, 2001.

[27] H. Zhu, P. A. V. Hall, and J. H. R. May, "Software unit test coverage and adequacy," *ACM Computing Surveys*, vol. 29, pp. 366-427, 1997.