Available online at www.sciencedirect.com

ScienceDirect

Information and Software Technology 50 (2008) 641–655

INFORMATION AND SOFTWARE TECHNOLOGY

www.elsevier.com/locate/infsof

Heuristics-based infeasible path detection for dynamic test data generation

Minh Ngoc Ngo *, Hee Beng Kuan Tan

School of Electrical and Electronic Engineering, Nanyang Technological University, Block S2, Nanyang Avenue 639798, Singapore

Received 15 January 2007; received in revised form 14 June 2007; accepted 19 June 2007
Available online 29 August 2007

Abstract

Automated test data generation plays an important part in reducing the cost and increasing the reliability of software testing. However, a challenging problem in path-oriented test data generation is the existence of infeasible program paths, where considerable effort may be wasted in trying to generate input data to traverse the paths. In this paper, we propose a heuristics-based approach to infeasible path detection for dynamic test data generation. Our approach is based on the observation that many infeasible program paths exhibit some common properties. Through realizing these properties in execution traces collected during the test data generation process, infeasible paths can be detected early with high accuracy. Our experiments show that the proposed approach efficiently detects most of the infeasible paths with an average precision of 96.02% and a recall of 100% of all the cases.

Keywords: Infeasible path detection; Dynamic test data generation; Heuristic-based

1. Introduction

Software testing is an important stage which accounts for 50% of the cost of software development [24]. It provides methods to establish confidence in the reliability of software. An important problem that arises in structural (white-box) testing is the generation of test data that causes a program to exercise some components such as statements, branches, paths, etc. However, test data generation is an extremely complicated and time-consuming task as it requires a careful analysis and understanding of the source code and knowledge of the underlying concepts of the testing criterion. For this reason, automation of test data generation may provide a significant cost and time reduction for software testing.

Structural based test data generation strategies are generally classified as path-oriented and goal-oriented. In path-oriented test data generation, a set of paths is selected which satisfy one or more coverage criteria and test data are generated to exercise these paths. In goal-oriented test data generation, test data are generated to execute some components (statements, branches, data flow, etc.) regardless of the path taken.

A challenging problem with path-oriented test data generation in particular and path-based software analysis in general is the existence of infeasible paths for which there is no input data for them to be executed. In this case, considerable effort might be wasted in trying to generate data for infeasible paths. Identification of infeasible paths in a program is an undecidable question [18]. To the best of our knowledge, no approach has successfully identified a large number of infeasible program paths. Existing work on infeasible path identification are based mainly on symbolic evaluation [42], control and data flow or static branch correlation [3,10,26] to provide partial solutions for some particular cases.

In this paper, we propose an approach for infeasible path detection, which can be integrated with any dynamic path-oriented test data generation technique [5,8,16,17,24,29,30,41] to enhance the performance of these techniques

* Corresponding author.
E-mail addresses: ngom0002@ntu.edu.sg (M.N. Ngo), ibktan@ntu.edu.sg (Hee Beng Kuan Tan).

0950-5849/$ - see front matter © 2007 Elsevier B.V. All rights reserved.
doi:10.1016/j.infsof.2007.06.006

in the presence of infeasible paths. Dynamic techniques are based on the program's real execution where actual values are assigned to input variables and the program's execution flow is monitored. If the program's execution deviated from the intended path, techniques such as function optimization, genetic algorithms or iterative relaxation are used to determine new values for the input variables which make the right branch to be taken.

Although it is impossible to solve the general problem of identifying all infeasible paths, we have discovered through empirical study that most of the infeasible paths follow some common properties, we called them *empirical properties*. During the test data generation process for a path, the program execution is monitored and checked against the empirical properties. If a match occurs, the path is highly infeasible and the test data generation process is terminated immediately without any wasted effort. In opposite to those approaches which use genetic algorithms or metrics to guide the identification of likely infeasible paths [4,27], our approach is based on empirical properties; thus, it is simple, yet fast and effective.

The rest of the paper is organized as follows. Section 2 defines and reviews some terminologies that will be used throughout this paper. Section 3 presents a theory for infeasible path detection based on empirical properties. Section 4 shows how the proposed theory can be used to detect infeasible path during dynamic test data generation. Section 5 describes the prototype system implemented for Java using evolutionary test data generation. Section 6 reports our experimental results. Section 7 accesses threats to validity. Section 8 discusses related work and Section 9 concludes the paper.

2. Terminology

In this section, we define and review some technical terms, which will be used throughout this paper.

In this paper, we use the term **program** to refer to any software unit such as function, method or procedure.

The structure of a program is represented by a direct graph called **Control Flow Graph (CFG)** $G = (N, E,$ 'begin', 'end') where N is a set of nodes, E is a set of edges, 'begin' is the unique entry node and 'end' is the unique exit node. Each node represents a program statement. Each edge (s,t) represents the flow of control from statement s to statement t. For a program with multiple exit points, we introduce an artificial 'end' node and insert a control flow edge from a real 'end' node to the artificial 'end' node.

To make our discussion throughout this paper concrete, we use a simplified `compute_grade` program, whose pseudo-code and CFG are given in Figs. 1 and 2, respectively. This program computes the grade of a student based on the mark. Finally, it updates the record of the student in the database based on `studentID`.

A path is a node sequence $\Omega = (x_1, x_2, \ldots, x_k), k \geqslant 2$, such that an edge exists from x_i to x_{i+1}, $1 \leqslant i \leqslant (k-1)$, i.e., $(x_i, x_{i+1}) \in E$. A path in a CFG that starts at the 'begin' node and ends at the 'end' node is called a **program path**.

```
procedure compute_grade(int mark, int studentID)
begin
1.        grade = “”
2.        if (mark ≤ 100) {
3.             if (mark < 50)
4.                  grade = “Fail”
               else
5.                  grade = “Pass”
          }
6.        if (grade != “”)
7.             update the student with ‘studentID’ record in the database
end
```

Fig. 1. compute_grade program.

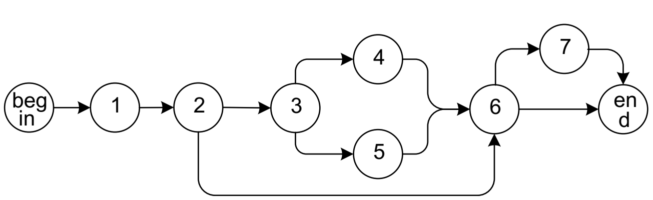


Fig. 2. CFG of the compute_grade.

Let v and w be nodes in a CFG such that $w \neq v$. We say v **post-dominates** w $(w \neq v)$ in the CFG if every directed path from w to the end node has to pass through v.

Let x and y be nodes in a CFG. Node y is **control-dependent** [9] on node x if and only if:

- There exists a directed path Ω from x to y such that except x and y, y post-dominates each node z in Ω, and
- y does not post-dominate x.

Furthermore, we say that node y is **transitively control-dependent** on node x if there exists a sequence $(z_0 = x, z_1, z_2, \ldots, z_n = y)$ of nodes in the CFG such that $n \geqslant 1$ and z_j is control-dependent on z_{j-1} for all j, $1 \leqslant j \leqslant n$. For example, In Fig. 2, node 4 is transitively control-dependent on node 2 because node 4 is control-dependent on node 3 and node 3 is control-dependent on node 2.

In this paper, we shall use the term **variable** in a general sense to refer to any variable, attribute of object (instance variable) or return value of method or function-call.

A **simple predicate** is a Boolean variable or a relational expression possible with one or more NOT (¬) operators. From predicate calculus, any predicate can be always be transformed into conjunctive normal form. From this fact, it is not difficult to verify that any predicate can always be transformed into a sequence of nested predicates such that each of these predicates is a simple predicate. Therefore, without loss of generality, in this paper, unless otherwise stated, we will assume that the predicate at any conditional statement is a simple predicate.

In a program, a variable **influences** a predicate if the predicate references to the variable. Let V be a variable used by a conditional statement x and let y be a statement which defines V where there is a path in the CFG from y to

x such that V is not redefined. Variable W **influences** variable V used in x if y references to W or is control-dependent on a node that references to W.

For example, variable `mark` used in statement 3 influences variable `grade` used in statement 6. This is because variable `grade` is defined in statement 4 which is control-dependent on statement 3 and along the path (3,4,6), variable `grade` is not redefined.

If variable U influences variable V and V influences variable W, then U also influences W. If variable W influences variable V and V influences a predicate, then W also influences the predicate.

Let $\Im(p)$ be a set of variables in a program (for example, a method or a function) such that for each variable V in $\Im(p)$:

- V influences predicate p
- there exists no variable which influences both V and p

then $\Im(p)$ is called **the minimal set of variables in the program that influence the predicate**.

As an example, consider predicate p = (`grade != ''`) in statement 6 in the pseudo-code in Fig. 1. Variable `grade` influences p. Variable `grade` is influenced by variable `mark` used in statement 3. Therefore, variable `mark` also influences p. However, the minimal set of variables influence p contains only variable `mark`. Variable `grade` is not included in the set because variable `mark` influences both variable `grade` and p.

An **input variable** ω_i of a program P is a variable which appears in an input statement or an input parameter. $Input(P) = (\omega_1, \omega_2, \ldots, \omega_n)$ is an array of input variables of the program P.

The **domain** $D\omega_i$ of the input variable ω_i is the set of values ω_i can hold. The input domain D of the program P is the product $D = D_{\omega_1} \times D_{\omega_2} \times \ldots \times D_{\omega_n}$, where $D\omega_i$ is the input domain of the variable ω_i. A point ω in the n-dimensional input space D, $\omega \in D$, is referred to as an input or test input.

A path is **feasible** (or executable) if there exists an input $\omega \in D$ for which the path is traversed during the program execution; otherwise, the path is **infeasible** (or unexecutable).

3. A theory for infeasible path detection

In this section, we formalize our findings on infeasible program paths in the form of empirical control flow graph properties. In the next section, we show how the theory can be applied to determine the infeasibility of a path during dynamic test data generation.

3.1. Cause of infeasible paths

It has been reported that the main cause of infeasible program paths is the correlation between some conditional statements along the path [3,40]. Two conditional statements are **correlated** if along some paths, the outcome of the later can be implied from the outcome of the earlier [3].

Conditional statements 2 and 6 in Fig. 1 are correlated because the outcome of the predicate q = (`grade != ''`) in statement 6 can be implied from the outcome of the predicate p = (`mark` $\leqslant$ `100`). More specifically, p is `true` implies that q is `true` and p is `false` implies that q is `false`.

In the presence of correlation, some program paths are not executable [3]. In the previous example, nodes 2 and 6 are correlated, as such, the following path is infeasible:

$$\Omega = (\text{begin}, 1, 2, 3, 4, 6, \text{end})$$

If following path Ω, when control reaches node 6, the outcome of the predicate q is always `true` because the outcome of predicate p is `true` (the 'true' out-coming edge of node 4 is not followed). Path Ω takes the 'false' out-coming edge of 6. In this case, there is no input on which Ω is traversed; thus, according to the definition, path Ω is infeasible.

Obviously, information about correlated conditional statements in a program is very useful in identifying infeasible program paths. For each pair of correlated conditional statements, there is at least one infeasible path [3]. Some of the correlations between conditional statements can be detected at compile time. However, it is reported by Bodik et al. [3] 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.

3.1.1. Empirical correlation

Theoretically, there is no rule which can tell whether two conditional statements are correlated. However, we have observed through the study of many programs that correlated condition statements do exhibit some common properties. Consequently, we introduce the concept of empirically correlated conditional statements. The main idea of introducing this concept is inspired by the difficulties in identifying correlated conditional statements statically. More importantly, during our empirical study, we also discover that there exist some relationships between empirically correlated statements and the infeasibility of the paths which pass through these statements.

Let x and y be two conditional statements in a program and let p and q be the predicate expressions of x and y, respectively. Let $\Im(p)$ and $\Im(q)$ be the minimal sets of variables that influence p and q, respectively. We say that x and y are **empirically correlated**(**e-correlated**) if the following conditions hold:

1. There exists a path in the control flow graph from x to y.
2. x is not transitively control-dependent on y and y is also not transitively control-dependent on x.
3. $\Im(p) = \Im(q)$.

If two conditional statements x and y are e-correlated, the pair $<x,y>$ is called an **e-correlated pair**.

Informally, two statements x and y are e-correlated if there is a path from x to y in the CFG, x and y must not have any control dependency relationship and they must be influenced by the same minimal set of variables. Clearly, the empirical correlation between two conditional statements does not depend on the distance (number of statements) between them in a path or a program. The empirical correlation is purely dependent on the control and data dependency between the two conditional statements.

According to this definition, in the program in Fig. 1, conditional statements at nodes 2 and 6 are e-correlated because:

1. There exists path $\Omega = (2,3,4,6)$ connecting the two nodes.
2. They do not have any control dependency relationship.
3. The set of minimal variables influence the predicates of the two nodes are identical and equal to {`mark`}.

Even though node 2 and 3 are influenced by the same set of variables, {`mark`}, node 3 is control-dependent on node 2; therefore, they are not e-correlated.

Empirically, we discover that if two conditional statements in a program are e-correlated, then their predicate expressions are either equivalent or mutually exclusive. This is formalized in the first empirical property.

Empirical Property 1. *Let x and y be two conditional statements in a program with predicate expressions p and q respectively. If x and y are e-correlated, then probably one and only one of the following expressions hold*:

1. $(p \rightarrow q) \wedge (\neg p \rightarrow \neg q)$
2. $(p \rightarrow \neg q) \wedge (\neg p \rightarrow q)$

If p and q satisfy the first condition, we say that p and q are **mutually exclusive**, *otherwise, they are* **equivalent**.

The first condition of Empirical Property 1 implies that there are two feasible paths, one passes through the ‘true’ branch of x and the ‘true’ branch of y; the other one passes through the ‘false’ branch of x and the ‘false’ branch of y.

Similarly, the second condition of Empirical Property 1 implies that there are two feasible paths, one passes through the ‘true’ branch of x and the ‘false’ branch of y; the other one passes through the ‘false’ branch of x and the ‘true’ branch of y.

For example, node 2 and node 6 in Fig. 2 are e-correlated with predicate expressions $p = ($`mark` $\leqslant$ `100`$)$ and $q = ($`grade != ‘‘’’`$)$ respectively. It is easy to verify that:

- The feasible path (begin, 1, 2, 3, 6, 7, end) passes through the ‘true’ branch of node 2 and the ‘true’ branch of node 6.
- The feasible path (begin, 1, 2, 6, end) passes through the ‘false’ branch of node 6 and the ‘false’ branch of node 8

Therefore, p and q satisfy the first condition of Empirical Property 1; thus they are equivalent. It is also easy to verify that p and q are not mutually exclusive.

3.1.2. Validation of Empirical Property 1

We conduct experiments to perform a binomial test to validate Empirical Property 1. The sample for validation was randomly collected from the following six Java systems:

1. AVIEC: This is an automated tool for testing and verifying the input error correction features for PHP applications.
2. Taxier: This is an automated taxi dispatching system. The system also includes a real-time simulation module to simulate the result of the dispatching algorithm used.
3. HtmlParser [19]: This is a HTML parser.
4. Jhotdraw [21]: This is a well-known and powerful GUI framework for technical and structured graphics. The framework has been developed following design patterns.
5. JlibSys [22]: This is a java library management system which is designed to simplify the task of managing a physical library consisting of various inventory items.
6. NCS: This is a commercial web-based health care system. For confidential reason, we do not provide detailed descriptions of the product here.

All the selected systems are written in Java. To ensure the generality of the proposed approach, all the systems are of different nature. Also, we collect sample from different sources: AVIEC is a research prototyping developed by postgraduate students; Taxier is a final year undergraduate project; NCS is an industrial system and the rest are open sourcesystem downloaded from Sourceforge [35].

For the validation of Empirical Property 1, each pair of e-correlated conditional statements forms a **case** for testing the empirical property. The alternate hypothesis states that Empirical Property 1 holds for equal or more than 90% of the cases. And, therefore, the null hypothesis states that Empirical Property 1 holds for less than 90% of the cases. That is:

$$H_0(\text{null hypothesis}): p(\text{Empirical Property 1 holds}) < 0.9$$

$$H_1(\text{alternate hypothesis}): p(\text{Empirical Property 1 holds}) \geqslant 0.9$$

The binomial test statistics z is computed as follows:

$$z = \frac{X/n - p}{\sqrt{(p(1-p)/n)}}$$

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 0.0005 as the type I error, if $z > 3.29$, we reject the null hypothesis.

We randomly pick some methods from each system for the experiment. We do not choose all methods from each application because that would form a huge number of methods for the hypothesis testing. Our objective is to collect a sample with reasonable size, yet containing cases from systems in different application domain. The number

Table 1
Hypothesis testing of Empirical Property 1

System	#Method	#Cond	#e-corr	#e-corr$_S$
AVIEC	217	314	114	102
Taxier	42	415	155	152
Htmlparser	513	891	103	101
Jhotdraw	17	98	54	51
JlibSys	69	221	93	90
NCS	342	574	234	215
Total	1200	2513	753	711

of method selected from each system and the total number of conditional statements in all the selected methods are listed in columns *#method* and *#cond* in Table 1 respectively.

We have developed a prototype system called jTGEN^I, whose architecture is depicted in Fig. 5, for the validation and evaluation of the proposed approach. The description of jTGEN^I will be presented in Section 5.

For each procedure, jTGEN^I computes all pairs of e-correlated conditional statements based on the control and data dependency information. For each pair of e-correlated statements <x,y>, jTGEN^I presents a decomposition slice [11] of the respective program containing all the statements which affect the computations of x and y. We then manually inspect the decomposition slice to determine whether the predicate expressions of x and are equivalent or mutually exclusive. If they are equivalent (mutually exclusive), we carefully study the source code to ensure that they cannot be mutually exclusive (equivalent). If this is confirmatory, we conclude that Empirical Property 1 holds for the case.

Results of this experiment are shown in Table 1. The columns #e-*corr*, #e-*corr*$_S$ and give the number of e-correlated pairs computed for each system and the number of e-correlated pairs computed which satisfy Empirical Property 1, respectively. As shown in Table 1, the total number of cases for testing Empirical Property 1 is 753. There are 42 cases which violate Empirical Property 1; thus, there are 711 cases which support the empirical property, the z score is 4.04 ($X = 711, n = 753, p = 0.9$). Since the z score is greater than 3.29, we reject the null hypothesis and conclude that Empirical Property 1 holds for more than 90% of all the cases at 0.05% level of significance.

3.2. Properties of infeasible paths

We observe that in the presence of e-correlated statements, some program paths are not executable. This property is presented next.

Empirical Property 2. *Let x and y be two conditional statements in a program. If x and y are e-correlated then there exists at least one infeasible path which passes through both x and y.*

Proof. Without loss of generality, we assume that each conditional statement has two out-coming branches ('true' and 'false' branch). Let (x, x_t) and (x, x_f) be the 'true' and 'false' out-coming branches of x, respectively. Similarly, let (y, y_t) and (y, y_f) be the 'true' and 'false' out-coming branches of y, respectively. Assume that all the paths passing through x and y are feasible. Consider the following four paths:

$$\Omega_1 = (\text{begin} \quad x, x_t \quad y, y_t \quad \text{end})$$
$$\Omega_2 = (\text{begin} \quad x, x_t \quad y, y_f \quad \text{end})$$
$$\Omega_3 = (\text{begin} \quad x, x_f \quad y, y_t \quad \text{end})$$
$$\Omega_4 = (\text{begin} \quad x, x_f \quad y, y_f \quad \text{end})$$

Let p and q be the predicate expressions of x and y respectively, from Ω_1 and Ω_4, it is inferred that $(p \to q \wedge \neg p \to \neg q)$ is true; thus p and q are equivalent. However, from Ω_2 and Ω_3, it is inferred that $(p \to \neg q \wedge \neg p \to q)$ is true; thus p and q are also mutually exclusive. This violates Empirical Property 1. Therefore, there must exist at least one infeasible path which passes through both x and y □.

According to Empirical Property 2, there is at least one infeasible path between any e-correlated pair. However, not all paths which go through a pair of e-correlated statements are infeasible. For example, consider the CFG in Fig. 2, there are six program paths in total, all of which contain the two e-correlated nodes 2 and 6. However, only three paths are infeasible. They are: (begin, 1, 2, 3, 4, 6, end), (begin, 1, 2, 3, 5, 6, end) and (begin, 1, 2, 6, 7, end). It is essential that we can identify which paths through an e-correlated pair are infeasible.

Static analysis techniques can be used to compute all the e-correlated conditional statements based on control and data dependency information between statements. However, for a given e-correlated pair <x,y>, static analysis cannot tell whether the predicate expressions of x and y satisfy the first or the second condition of Empirical Property 1. This is because the actual values of these predicate expressions are not known during static analysis. Consequentially, static analysis cannot identify the exact infeasible paths through x and y. Dynamic information can solve this problem.

Dynamic information is precise because it is collected during the execution of a program, which reflects the real behaviors. An **execution trace** of a program is actually a sequential list of all the statements which are invoked during the execution of the program. Each execution trace, therefore, corresponds to a feasible program path.

We discover that if we can collect an execution trace containing an e-correlated pair of conditional statements x and y, we will be able to identify the infeasible paths passing through x and y. This observation is formalized in the next empirical property.

Empirical Property 3. *Let x and y be two e-correlated conditional statements in a program. It is highly probable that the following statements hold*:

1. If there exists an execution trace which passes through the 'true' out-coming branch of x and the 'true' ('false') out-coming branch of y, then any path which passes through the 'true' out-coming branch of x and the 'false' ('true') out-coming branch of y is infeasible.

2. If there exists an execution trace which passes through the ‘false’ out-coming branch of x and the ‘true’ (‘false’) out-coming branch of y, then any path which passes through the ‘false’ out-coming branch of x and the ‘false’ (‘true’) out-coming branch of y is infeasible.

Proof. Let (x, x_t) and (x, x_f) be the ‘true’ and ‘false’ out-coming branches of x respectively. Similarly, let (y, y_t) and (y, y_f) be the ‘true’ and ‘false’ out-coming branches of y respectively. Let p and q be the predicate expressions of x and y respectively; since x and y are e-correlated, according to Empirical Property 1, p and q must be equivalent or mutually exclusive. We now prove that the first statement holds. If there exists an execution trace which contains (x, x_t) and (y, y_t), ‘$p \rightarrow q$’ is true. Consider the path $\Omega =$ (begin x, x_t y, y_f end) which passes through the ‘true’ branch of x and the ‘false’ branch of y. If Ω is feasible, ‘$p \rightarrow \neg q$’ is true; and Empirical Property 1 is violated. Therefore, Ω must be infeasible. Similarly, if there exists a feasible path which contains (x, x_t) and (y, y_f), we can prove that a path which passes through the ‘true’ branch of x and the ‘true’ branch of y is infeasible. The proof for the second statement is similar. □

As an illustration, node 2 and 6 in the control flow graph in Fig. 2 are e-correlated with $p =$ (mark $\leqslant$ 100) and $q =$ (grade != ‘‘’’). Let $E =$ (begin, 1, 2, 3, 4, 6, 7, end) be an execution trace with input mark = 40 and studentID = 10. E contains the ‘true’ branch of node 2 and the ‘true’ branch of node 6; thus, according to Empirical Property 3, any path which passes through the ‘true’ branch of node 2 and the ‘false’ branch of node 6 is infeasible. Indeed, it is easy to verify that the following paths are infeasible:

- (begin, 1, 2, 3, 4, 6, end)
- (begin, 1, 2, 3, 5, 6, end)

4. Test data generation

Let $\Omega = (x_1, x_2, \ldots, x_n)$ be a program path, the main goal of the path-oriented test data generation problem is to find an input $\omega \in D$ on which Ω will be traversed. Our approach improves dynamic path-oriented test data generation techniques [16,17,24,25,43] by introducing a new step to detect infeasible path based on the empirical properties introduced in the previous section. In other words, our approach augments dynamic test date generation techniques to improve the performance of these techniques in the presence of infeasible path. Empirical Property 3 plays an important part in the proposed approach because it relates e-correlated pairs with the infeasibility of program paths containing the pair through execution traces collected during test data generation.

For the ease of presentation, we choose the test data generation algorithm proposed by Korel in [24] to present in this section. Since the algorithm is straightforward, we can focus on describing the augmentation of infeasible path detection with the test data generation process. However, the infeasible path detection step can be integrated with any dynamic test data generation technique. Fig. 4 is the flowchart of approach to test data generation with infeasible path detection presented in this section.

Without loss of generality, we can assume that the relational expression at each simple predicate is a simple relational expression. That is, all predicates can be represented as

$$exp_1 \ \boldsymbol{op} \ exp_2$$

where exp_1 and exp_2 are arithmetic expressions and op is one of $\{<, \leqslant, >, \geqslant, =, \neq\}$. Therefore, each predicate can be transformed into the equivalent predicate of the form

$$F \ rel \ 0$$

where F and rel are given in Table 2. We refer to F as *predicate function* which is positive when the predicate is false or negative when the predicate is true.

The problem of finding an input to execute Ω can be reduced to a sequence of sub-goals where “each sub-goal is solved using the function minimization technique” [24]. Our approach starts with an arbitrarily chosen input. Let ω^0 be the initial input of the program. If Ω is traversed, ω^0 is the solution of the test data generation problem; otherwise, refine the initial input.

Algorithm isInfeasible (Ω, Ψ) (Fig. 3) determines whether the desired path Ω is infeasible given the execution trace generated by generated by ω^0 is Ψ. Let $\Omega_0 = (x_1, x_2, \ldots, x_k)$ be the longest sub-path of traversed on ω^0 (line 1), the branch violation occurs on the execution of branch (x_k, x_{k+1}). Definitely, Ψ follows the other out-coming branch of x_k, denoted as (x_k, x_{k+1}). The paths Ψ and Ω contain a common sub-path, which is Ω_0. We check whether Ω_0

Table 2
Predicate function

Predicate expression	Predicate function F	rel
$E_1 > E_2$	$E_2 - E_1$	$<$
$E_1 \geqslant E_2$	$E_2 - E_1$	$\leqslant$
$E_1 < E_2$	$E_1 - E_2$	$<$
$E_1 \leqslant E_2$	$E_1 - E_2$	$\leqslant$
$E_1 = E_2$	$\lvert E_1 - E_2 \rvert$	$=$
$E_1 \neq E_2$	$\lvert E_1 - E_2 \rvert$	$\leqslant$

```
Algorithm isInfeasible($\Omega$,$\Psi$)
Input   $\Omega = (x1, x2,\ldots,xn)$: the desired path
        $\Psi$: an execution trace where the branch violation occurs at $(x_k, x_{k+1})$
Output  'true' if $\Omega$ is infeasible; 'false' otherwise
Begin
  1.  $\Omega_0 = (x_1, x_2,\ldots,x_k)$
  2.  if ($\Omega_0$ contains an e-correlated pair $<x_k, x_l>$ where $1 \le l < k$) then
  3.        return true
  4.  return false
end
```

Fig. 3. Infeasible path detection algorithm.

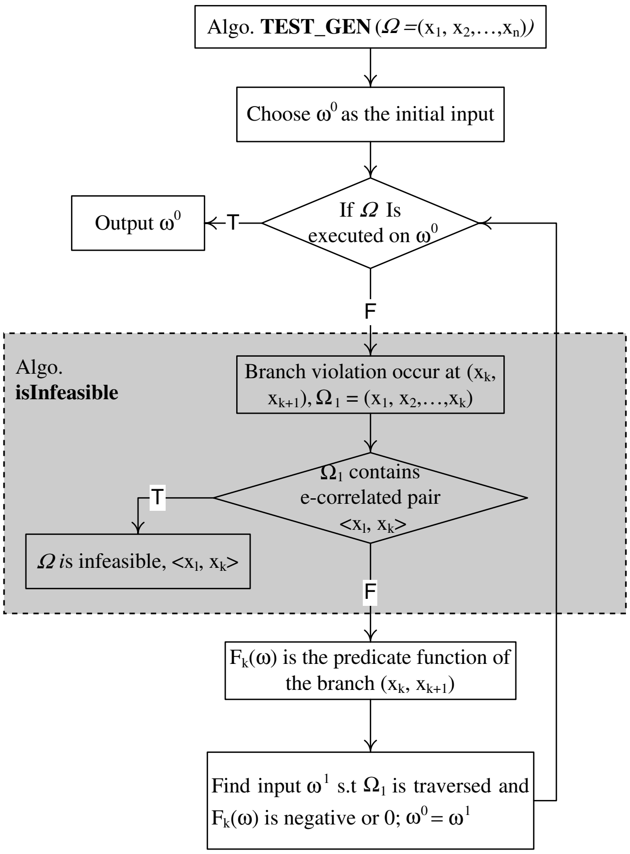

Algo. TEST_GEN (Ω =(x1, x2,…,xn))
Choose ω0 as the initial input
If Ω Is executed on ω0
T
Output ω0
F
Algo. isInfeasible
Branch violation occur at (xk, xk+1), Ω1 = (x1, x2,…,xk)
Ω1 contains e-correlated pair <xl, xk>
T
Ω is infeasible, <xl, xk>
F
Fk(ω) is the predicate function of the branch (xk, xk+1)
Find input ω1 s.t Ω1 is traversed and Fk(ω) is negative or 0; ω0 = ω1

Fig. 4. Flow chart of the proposed approach.

contains any e-correlated statement. If Ω_0 contains an e-correlated pair $<x_k,x_l>$ where $1 \leqslant l < k$, we conclude that Ω is an infeasible path (line 2, 3). Indeed, we have:

- $\Psi = (x_1,\ldots,x_l,x_{l+1},\ldots,x_k,x_{k+1},\ldots x_n)$ is an execution trace and
- $<x_k,x_l>$ is an e-correlated pair.

Therefore, according to Empirical Property 3, $\Omega = (x_1,\ldots,x_l,x_{l+1},\ldots,x_k,x_{k+1},\ldots,x_n)$ is an infeasible path. In this case, the test data generation process immediately terminates; otherwise, we solve the first sub-goal.

Let $F_k(\omega)$ be the predicate function of the branch (x_k,x_{k+1}), where a branch violation occurs during the execution of the program on ω_0. The first sub-goal now is to find an input which will still cause the execution of Ω_0 and at the same time cause $F_k(\omega)$ to be negative (or zero) at x_k. If such an input can be found, the branch (x_k,x_{k+1}) will be successfully executed. More formally, we want to find an input $\omega \in D$ satisfying

$$F_k(\omega)\ rel\ 0$$

subject to the constraint:

$$\Omega_0 \text{ is traversed on } \omega,$$

where $rel \in \{\leqslant,<,=\}$.

As the path $(x_1,\ldots,x_k,x_{k+1})$ is not likely an infeasible path, any numerical technique for constraint minimization [1] can be used to minimize $F_k(\omega)$ until it becomes negative with respect to the constraint. If the infeasible path detection scheme is not conducted before this step, in the case of infeasible path, a large number of iterations can be performed before the search procedure terminates and a lot of effort can be wasted.

The description of the search procedure is outside the scope of this paper. Next we only briefly describe the search procedure introduced by Korel [24]. The local search procedure is known as the *alternating variable method* for deriving input values to minimize the predicate function. In this method, each input variable is selected in turn and its value is adjusted while keeping other variable values constant. The search procedure starts with the *exploratory phase*, where the selected variable is increased and decreased by small amounts. If either move produces a smaller (improved) value of the predicate function while not violating the constraint, then a *pattern phase* is entered. In the pattern phase, a larger move is made in the direction of the improvement. A series of similar moves is made until the predicate function becomes negative. If this is the case, the next input variable is then selected for the exploratory phase.

The above search procedure, like any local search method, has the potential to make moves through variable values that cannot lead to an improvement in the value of the 19 predicate function. In [24], Korel introduces two techniques to improve the efficiency of the search procedure including influence graph and risk analysis. An influence graph is constructed based on dynamic dataflow analysis and is used to detect which input variables influence the outcome of the predicate at the conditional node where the branch violation occurs. The risk analysis of input variable identifies if they could potentially violate the already successful sub-path.

Let ω^1 be the solution of the firs sub-goal, the same process applied on ω^0 is repeated on ω^1. Moreover, this process of solving sub-goals is repeated until the solution to the main goal is found or a time-limit has passed.

Example. We walk through a simple example based on the program in Fig. 1 to illustrate our approach to test data generation with the augmentation of infeasible path detection. We apply the proposed approach to generate the test data for the following path in the CFG in Fig. 2: $\Omega = (\text{begin}, 1, 2, 3, 5, 6, \text{end})$.

First iteration

Initially, a random input is hosen: $\omega^0 = (200, 10)$, which means `mark` = 200 and `studentID` = 10. The program is executed with ω^0 and the corresponding execution trace is (begin, 1, 2, 6, end). As such, the branch violation occurs during the execution of the branch $(2,3)$ in Ω. The longest sub-path of Ω executed on ω^0 is $\Omega_0 = (\text{begin}, 1, 2)$. As Ω_0 does not contain any e-correlated pair, we solve the first sub-goal. The predicate function of the branch $(2,3)$ is $F_0 = (\texttt{mark} - 100)$. We want to find an input ω^1 such that F_0 is negative (or equal to zero), subject to the constraint that Ω_0 is executed. Such an input can be easily found using any functional minimization technique; assume that $\omega^1 = (40, 10)$ is the solution for the first sub-goal.

Second iteration

The execution trace of the program on ω^1 is (begin, 1, 2, 3, 4, 6, 7, end). As such, the path Ω is not followed and the branch violation occurs at $(3, 5)$. The longest sub-path of Ω executed on ω^1 is $\Omega_1 =$ (begin, 1, 2, 3). As Ω_1 does not contain any e-correlated pair, we solve the second sub-goal. The predicate function of the branch $(3, 5)$ is $F_1 = (50 - \texttt{mark})$. The second sub-goal is find an input ω^2 such that F_1 is negative (or equal to zero) and the path Ω_1 is followed. Assume that the solution of the second sub-goal is $\omega^2 = (60, 10)$.

Third iteration

The path executed on ω^2 is (begin, 1, 2, 3, 5, 6, 7, end). As such, the branch violation occurs during the execution of the branch (6, end) of the path Ω. The longest sub-path of Ω executed on ω^2 is $\Omega_2 =$ (begin, 1, 2, 3, 5, 6). As Ω_2 contains the e-correlated pair <2, 6>, the test data generation terminates with the conclusion that the path Ω is infeasible.

Time complexity. In the case of feasible path, the time complexity of the proposed approach is dominated by the algorithm used for dynamic test data generation. In the case of infeasible path, the test data generation process terminates as soon as a branch violation occurs at an e-correlated conditional statement. As such, in the worst case, the time complexity is still equal to the time taken by the test data generation process.

5. The prototype system

We have implemented a semi-automated path-oriented test data generation tool for Java called jTGEN^I (*Test data generator* for Java, the ^{I} stands for "with infeasible path detection integrated"). As the approaches proposed in [8,24] only deal with procedural programs, it is not suitable for Java in particular and object-oriented language in general. This is because "test cases for class methods must also account for the state of the objects on which method execution is issued" [36]. Therefore, we adopt the evolutionary test data generation technique proposed by Tonella [36] to generate test cases for a predefined path. Genetic Algorithms (GA) are probably the most well-known form of Evolutionary algorithms, which have been applied successfully to the problem of test data generation [38]. Basically, a population of 'individuals', representing test cases, is evolved based on the mechanics of natural selection and genetics in order to increase a measure of fitness, accounting for the similarities between the execution traces produced by these test cases and the desired path.

Note that the infeasible path detection algorithm can be intra-procedural or interprocedural depending on the control/data flow analysis algorithms and the test data generation technique. The current version of jTGEN^I only implements intra-procedural analysis algorithms. The architectural diagram of jTGEN^I is given in Fig. 5.

5.1. Soot

Soot [34] is an excellent program analysis tool which provides information on method signatures, call/control/data dependencies used by other sub-systems of jTGEN^I.

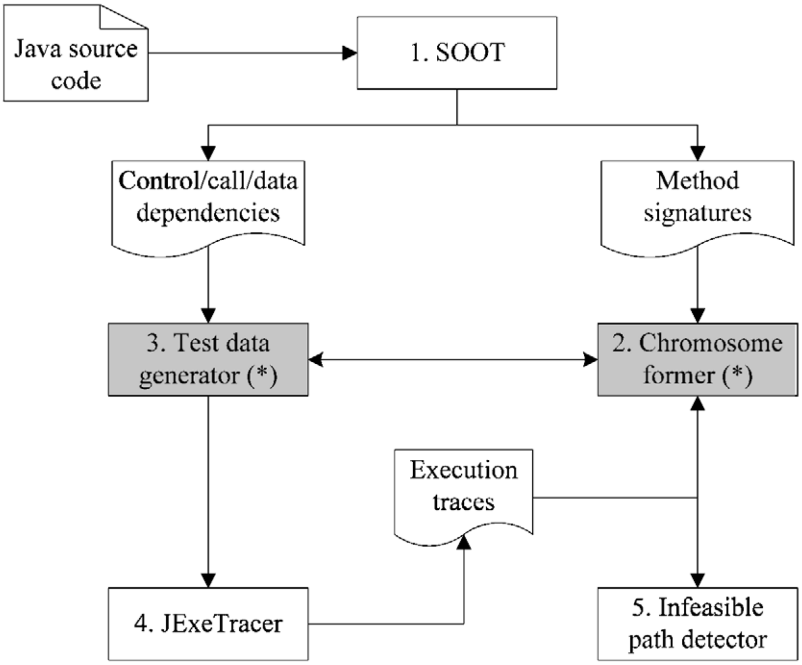


Fig. 5. Architectural diagram of jTGEN^I.

During the static analysis of a method, we use the three address code immediate representation called Jimple. We also use Soot's point-to analysis to:

- Identify the heap objects to which a reference variable can point.
- Resolve the potential destination of all polymorphic method dispatches.
- Create the interprocedural control flow graph.

5.2. Chromosome former

Chromosome former is adopted from the tool eToc (evolutionary testing of classes) developed by Tonella [36]. This sub-system constructs new chromosomes and mutates existing based on the information about method signatures. Basically, each chromosome encodes a sequence of statements for object creation, state change and method invocation. In addition, a chromosome also contains information which suggests some initial input values.

The two critical factors of the *Chromosome former* are the fitness function and the mutation operators. All the mutation operators proposed by Tonella [36] are kept unchanged. However, we use a different fitness function. While the fitness function implemented in eToc [36] aims to generate test cases for branch coverage or data flow testing (goal-oriented), we want to generate test case to force the execution of a path through a desired path (path-oriented). As such, we adopt the fitness function introduced by Bueno et al. [4] as follow:

$$F_t = NC - \left(\frac{EP}{MEP}\right)$$

NC is the path similarity metric which reflects the similarity between an execution trace and the desired path. *NC* is equal to the number of coincident nodes between the execution trace and the desired path, from the begin node of the method to the node where there is a branch violation. *EP* is the absolute value of Korel's predicate function [24] of the function where the branch violation occurs, as presented in the previous section. This value reflects the error that causes the executed path to deviate from the desired one. Lastly, *MEP* is the predicate function maximum value among the candidate solution that execution the same nodes of the desired path. Intuitively, the greater the number of correctly executed nodes, the greater is the fitness value of an individual. For those execution traces which the same number of correct nodes, the most adequate are those with smallest absolute value of predicate function where the branch violation occurs.

If the desired path is already in the set of execution traces produced by the current population, the test data generation process stops; otherwise, the test cases associated to the execution traces with higher similarity are recovered first.

5.3. Test data generator

The *Test data generator* sub-system is also adopted from the one implemented in eToc [36]. Test data for primitive data types are generated by the default generator, which implements the following rules:

- *Integer and real numbers:* Integer and real number are uniformly chosen in the interval $[0, 100]$.
- *Boolean:* Boolean values true and false are randomly chosen with equal probability (0.5).
- *Strings:* Characters inserted into the generated string are uniformly chosen among the alphanumeric characters ([a-zA-Z0-9]). The probability of inserting another character after inserting n characters is 0.5^{n+1}.

Inputs for non-primitive data structures are either generated manually (by the tester) or by a customized data generator. A customized data generator is a class is provided by the tester and extends the default generator to create objects of complex types based on primitive types.

These initial input values are displayed to the tester. The tester can edit input values if any of them does not make sense, for instance file name or a person name in the database.

The *Test data generator* then converts the actions encoded in each chromosome into constructor and method invocation and then to JUnit test cases by using the Java Reflection API.

5.4. JExeTracer

JExeTracer combines JUnit and the Sun's Java Debugging Interface (JDI) to launch the execution of a JUnit test case to trace in a debug Java Virtual Machine (JVM) and notify an event handler whenever a statement is executed. The event handler checks if the statement is the "end" statement of the program and record the path through the program that has been exercised. JExeTracer also computes the predicate function at each conditional statement during runtime. This information is then used by the *Chromosome former* to calculate the fitness value for each execution trace.

5.5. Infeasible path detector

This sub-system detects the infeasibility of the desired path based on the execution traces generated by JExeTracer. Two infeasible path detection algorithms have been implemented including the proposed approach and the heuristic proposed by Bueno et al. [4]; thus, there are two execution modes defined for jTGEN^{I}:

- ***B-mode*** (Bueno et al. [4] algorithm): In this mode, all the execution traces are used to compute the population's best fitness. This value is monitored to predict the infeasibility of the desired path. When generating test data for a feasible path, a continual population's best fitness improvement can be observed. However, when the path is infeasible, a persistent lack of progress of this value results. Bueno et al. [4] propose the following heuristics to identify the lack of progress identification:

I. In each new generation, if the increase in the best fitness is lower than δ, a counter is incremented. If the counter reaches NL, apply rule II.

II. If the lack of progress accumulated through NL generations is lower than Δ then the desired path is infeasible.

NL value determines how persistent the lack of search progress needs to be alerted to the tester about the likely unfeasibility of the path. NL is computed for each path by the following equation:

$$NL = A + B \times (Np + Nv)$$

where A is a constant; Np is the number of nodes in the desired path; Nv is the number of program input variables. B is a parameter which controls the influence of Np and Nv in NL. In the current implementation of jTGEN$^{\mathrm{I}}$, the following values are being used for the parameters: $\delta = 0.30$; $\Delta = 0.20$; $A = 30$, $B = 3$.

- ***P-mode*** (Proposed approach): In this mode, for each execution trace Ψ, the algorithm isInfeasible (Fig. 3) is called on the desired path and Ψ to check the infeasibility of the desired path. If the desired path is infeasible, it will be detected even more effectively than using Korel's test data generation approach presented in the previous section because in this case, more execution traces are used.

6. Experiments

We have conducted an experiment to evaluate the performance of the proposed approach in detecting infeasible path during dynamic test data generation. We also compare the performance of the proposed approach with the infeasible path detection algorithm proposed by Bueno et al. [4], which is referred to as *Bueno approach* in the rest of this paper.

6.1. Experiment design

We randomly select a set of classes from the following four Java systems for the experiment:

- **PMD** [31]: a Java source code analyzer. It finds unused variables, empty catch blocks, unnecessary object creation, and so forth.
- **JMathLib** [23]: a library of mathematical functions designed to be used in evaluating complex expressions and displaying the results graphically. It can be used either interactively or to interpret script files.
- **GFP** [13]: a Free Open Source Personal Finance Manager in Java/Swing that can run on virtually any Operating System. It's designed to help people with little financial knowledge managing their finances.
- **SOOT** [34]: a Java bytecode optimization framework.

For each class, jTGEN$^{\mathrm{I}}$ chooses only methods which contains at least 20 lines of code (LOC) due to the fact that simple methods do not contain many infeasible paths. For each method we automatically generate a set of linearly independent program paths, it is also referred to as basis paths [28] by following the algorithm proposed in [32]. Table 3 gives the number of methods chosen from each system, the average LOC of all the methods and the number of basis paths generated from the chosen methods.

Altogether, there are 2419 basis paths generated from 190 methods chosen from the four systems. For each execution mode (*B-mode* and *P-mode*) of jTGEN$^{\mathrm{I}}$, we carry out the following two steps:

- *Step 1*: For each basis path, jTGEN$^{\mathrm{I}}$ generates a test case for the path or concludes that the path is infeasible. In the latter case, we carefully study the source code to confirm that the path is indeed infeasible. If it is confirmatory, jTGEN$^{\mathrm{I}}$ has correctly detected the infeasible path; otherwise Ω is counted as a false positive cases.
- *Step 2*: We measure the precision and recall of the information obtained in this execution mode of jTGEN$^{\mathrm{I}}$. Let I_{truly}, I_{detected} and I_{false} be the number of basis paths which are truly infeasible, the number of paths concluded infeasible by jTGEN$^{\mathrm{I}}$ and the number of false positive cases produced by jTGEN$^{\mathrm{I}}$, respectively. Adopting the definition of precision from Information Retrieval field, the precision of the information obtained by jTGEN$^{\mathrm{I}}$ in this execution mode with respect to a system is measured by the percentage of the number of infeasible paths correctly detected, which is computed as follow:

  $$\text{Precision} = \frac{I_{\text{detected}} - I_{\text{false}}}{I_{\text{detected}s}} \times 100\%$$

  The recall (completeness) measures the ratio between the number of infeasible paths correctly detected by the proposed approach and the actual number of infeasible paths in the set of basis paths generated.

  $$\text{Recall} = \frac{I_{\text{detected}} - I_{\text{false}}}{I_{\text{truely}}} \times 100\%$$

  We then evaluate the performance of this execution mode based on the traditional *f*-measure as follows:

  $$f\_\text{measure} = 2 \times \frac{\mathit{Precision} \times \mathit{Recall}}{\mathit{Precision} + \mathit{Recall}}$$

Table 3
Java systems for the experiment

System	Number of methods	AvgLOC	Number of basis paths
PMD	52	28	482
JMathLib	80	21	691
GFP	87	36	679
SOOT	71	27	567

Table 4
Experiment results

System	I_{truely}	*B-mode* (Bueno approach)					*P-mode* (Proposed approach)				
		I_{detected}	I_{false}	Recall (%)	Pre (%)	*f*	I_{detected}	I_{false}	Recall (%)	Pre (%)	*f*
PMD	213	248	35	100	85.9	0.924	217	4	100	98.2	0.991
JMathLib	354	375	21	100	94.4	0.971	263	9	100	97.5	0.987
GFP	379	436	57	100	86.9	0.930	396	17	100	95.7	0.978
SOOT	267	285	18	100	93.7	0.967	287	21	100	92.7	0.962

f-measure weights low values of precision and recall more heavily than higher values. It is high if both precision and recall are high.

6.2. Results

Table 4 shows the results of our experiments in two execution modes of jTGEN$^{\text{I}}$. As shown in columns *Recall* of both execution modes, both the proposed approach and the *Bueno approach* achieve a 100% recall in all cases. This indicates that there is no infeasible path which is not detected by both approaches.

The columns *Pre* and *f* show the precision and recall, respectively, of each approach. The results show that, the proposed approach gives an average precision of 96.02% and an average *f*-measure of 0.980 while the *Bueno approach* gives an average precision of 90.22% and an average 0.948. Using the proposed approach, the highest achievable precision score is 98.2% and the lowest precision score is 92.7%. Using the *Bueno approach*, the highest achievable precision score is 94.4% and the lowest one is 85.5%. The *Bueno approach* results in a total of 131 false positive cases, which account for 9.75% of all the infeasible paths detected. The proposed approach results in only 51 false positive cases which account for 4.03% of all the cases detected; thus the number of false positive cases produced by the proposed approach is 2.42 times lesser as compared with the *Bueno approach*.

The false positive cases produced by the *Bueno approach* and the proposed approach are, however, not the same. Most of the false positive cases produced by the *Bueno approach* are hard-to-solve predicates, which cause a persistent lack of search progress, causing the incorrect conclusion on the infeasibility of a path.

We also carefully investigate all the false positive cases produced by the proposed approach. Most of the false positive cases that we discover so far are similar to the case in Fig. 6 taken from SOOT.

There are four paths in total and there is only one infeasible path. When (dest != this & & dest != other), path $(1,2,3,4)$ is executed. When (dest == other), path $(1,3,4)$ is executed. When (dest == this), path $(1,3)$ is executed. Let p be the predicate (dest != this & & dest != other). Let q be the predicate (dest != other). Path $(1,2,3)$ is infeasible because when p is true, q is also true. However, when p is false, q might be true or false depending on the value of dest. As such, the following expression holds:

$$(p \rightarrow q) \wedge [(\neg p \rightarrow q) \vee (\neg p \rightarrow \neg q)] \tag{1}$$

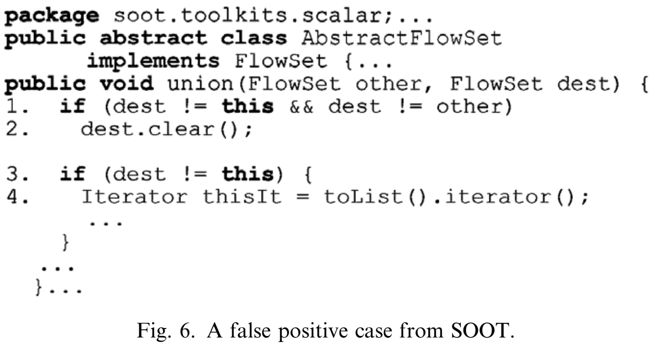
However, in this case, the proposed approach concludes that there are two infeasible paths. As such, one of the infeasible paths is false positive. Currently, our approach cannot handle this kind of code pattern. The logic behind this code pattern is that for those values of dest not equal to this, actions in line 4 (Fig. 6) will be executed. If dest is also not equal to other, in addition to actions in line 4, actions in line 2 will also be executed.

Another code pattern found among the false positive cases is illustrated in Fig. 7:

This code pattern is also taken from SOOT. Let p be the predicate expression (a instanceof classA). Let q be the predicate expression (a instanceof classB). Similar to the case in Fig. 6, when p is true, q must be false; however when p is false, q might be false or true depending on the value of the variable a. As such, the following expression holds:

$$(p \rightarrow \neg q) \wedge [(\neg p \rightarrow q) \vee (\neg p \rightarrow \neg q)] \tag{2}$$

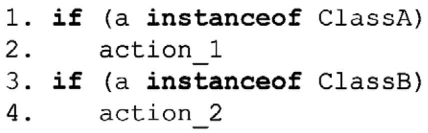
The expression $(\neg p \wedge \neg q)$ means that a is not the instance of both ClassA and ClassB. For this code pattern, there is also only one infeasible path, which is $(1,2,3,4)$. However,



```
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();
      ...
    }
  ...
  }...
```

Fig. 6. A false positive case from SOOT.



```
1. if (a instanceof ClassA)
2.    action_1
3. if (a instanceof ClassB)
4.    action_2
```

Fig. 7. A false positive code pattern from SOOT.

the proposed approach concludes that there are two infeasible paths; thus resulting in one false positive case.

6.3. Discussions

The proposed approach does not require any additional effort for the infeasible path detection. Conversely, the proposed approach enhances any dynamic test data generation approach in the sense that it saves much effort in the presence of infeasible paths. It simply augments a dynamic test data generation approach, uses the execution traces generated during the test data generation process and concludes on the infeasibility of the path under consideration by checking against the empirical properties.

The proposed approach occasionally makes wrong conclusion on the infeasibility of a path. Our empirical study shows that the code patterns found in those false positive cases are not frequent, only 4.03% of all the cases. This suggests that the proposed infeasible path detection technique is useful for structural testing practice as the current industry standard for coverage is in the range of 80–90% [12]. Currently, we are investigating the use of simple symbolic substitution to solve this problem. However, computational complexity is another problem that we need to solve when using symbolic substitution. Expressions (1) and (2) also serve as good suggestions to extend our current set of empirical properties to handle those false positive cases.

Nonetheless, it is important to note that there is no absolute solution to the problem of infeasible path. If the proposed approach concludes that a path is infeasible, it is highly probable that the path is truly infeasible. The tester can always have the choice to confirm the infeasibility by analyzing the code. In this case, the e-correlated pair which suspect of causing the infeasible path is very useful in his/her analysis.

7. Threat to validity

This study, like any other empirical study, has some limitations.

The binomial tests presented in Section 3.1.2 for validating Empirical Property 1 are influenced by our choice of sample sets and the subject applications. Even though we have randomly chosen systems from various application domains which are developed by software engineers of different level the chosen systems might not be the representatives of the population. However, we surmise that our conclusions will generalize since they are based on results that are consistent over a population of sample sets that are greatly varied in size.

The hypothesis testing of Empirical Property 1 was conducted on a large sample size in which each e-correlated pair produced by jTGEN^I was manually checked whether their predicate expressions are mutually exclusive or identical. Similarly, the experiment presented in Section 6 requires much manual effort in determining the infeasibility of a path. These are tedious and error-prone tasks. To better assist in the hypothesis testing of Empirical Property 1, we have integrated a decomposition slicer in jTGEN^I. This component produces a slice of a program containing all the statements involving in the computations of a pair of e-correlated conditional statements. The decomposition slice is useful in understanding the relationship of an e-correlated pair to identify whether they are mutually exclusive or identical. We have spent more than one year for the validation and evaluation of the proposed approach with one author working in pair with team of final year students to counter check any conclusion. Moreover, thanks to tools such as SOOT, GraphViz [15], Kaveri [20] and eToc, we are able to focus our analysis on a particular code segment, visualize the CFG of a method as well as data and call dependency information; thus the manual checking has been made much more productive.

8. Related work

8.1. Infeasible path detection

The causes and effects of infeasible program paths have been discussed early by Hedley and Hennel [18].

A metric to predict the feasibility of a program path and a test generation strategy is proposed by Malevris [26,27]. Both of them are based on the observation that the greater number of conditional statements contained within a path, the greater the probability of it to be infeasible. χ^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 base on empirical properties and use hypothesis testing to validate the correctness of the properties. However, the main difference between this technique and ours is that our approach combines heuristic with dynamic information to detect a large portion of infeasible paths.

Forgacs et al. [10] believe that the main goal of testers is generally not to execute a specific path but rather to reach one (or more selected program points where they think that it must be tested). Therefore, instead of minimizing the number of conditional statements on a path as having been suggested by Malevris [27], the number of predicates which are actually determinant to reach that point should be minimized. They 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 statistics are given to show the effectiveness of the proposed heuristic.

Bodik et al. [3] 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 [2], which shows that 9% to 40% of conditional statements in a large number of programs exhibit correlation which can be detected statically during compile time. However, it is reported in [3] 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.

Bueno et al. [5,4] present a tool which uses genetic algorithms to identify potentially infeasible program path during the test data generation. They propose a fitness function which combines both control flow and data flow information to better direct the search. However, only a small-scale experiment (40 infeasible paths) has been conducted to validate the correctness of the approach. Computational cost of infeasible path identification is another weakness of this approach.

Symbolic execution based approaches to infeasible path identification have been proposed in [10,42]. A path is represented by a set of constraints or a formula. The formula is solvable if and only if there are input values which drive the execution of the program down to the path. A theorem prover or a constraint solver is invoked to test the feasibility of the path. However, only standard data types such as integer, Booleans and array are accepted.

8.2. Test data generation

Dynamic test data generation techniques which are based on actual execution of the program under test have been investigated by many researchers [16,24,43]. This approach thus overcomes the difficulties in handling complex data structures. The basic idea is that the cost function associated with each branch predicate of a path is minimized such that the intended path is followed. In the cases of infeasible path, much effort will be wasted in searching for the test data to execute the path.

Gupta et al. [16,17] introduce the iterative relaxation method to test data generation. In each iteration, statements relevant to the evaluation of each branch predicate are executed and a set of linear constraints are derived using the relaxation techniques. If all the branch predicates along the path are linear, the approach generates test data in just one iteration or concludes that the path is infeasible.

Genetic algorithms have been applied successfully to test data generation problems [38]. Michael et al. [29] introduce the GADGET system which used genetic algorithms to perform the functional minimization during dynamic test data generation. The approach differs from those proposed in [17,25] in that genetic algorithms are less susceptible to local minimization, which can cause the test data generation process halt without finding any adequate input. Pargas et al. [30] use control dependency information to guide a genetic algorithm in searching for test data. Bueno et al. [5] present a tool for the automation of both test data generation and infeasible path identification using a fitness function which evolve the population such that the individuals get closer to the intended path. Tonella [36] applies genetic algorithm to the testing of classes. The author presents a format for chromosome which encodes a method sequence for object creation, state changes and initial input values.

Symbolic execution based test data generation [6,14,37,39,41] executes method sequences with symbolic input parameters, builds path constraints on parameters and solves the constraints to create actual input values. Check-n-Crash[7] creates abstract constraint over input variables that cause exceptional behaviors and use a constraint solver to derive concrete test inputs that exhibit the behaviors. Infeasible paths are generally detected if a path constrain is found to be inconsistent with some others. Like any method using symbolic execution, these approaches are expensive. Further more, they do not scale up well for complicated programs. Symbolic execution is also limited by the difficulties in handling loops, arrays, pointers and procedure calls.

Concolic testing removes the limitations of symbolic testing in that a hybrid mix of symbolic and concrete execution is used simultaneously to generate different inputs exercising distinct behaviors of a program. Concrete execution simplifies symbolic expression that cannot be solved by a constraint solver at the same time guides the symbolic execution along an intended path. Concolic test data generation has been successfully implemented in CUTE and jCUTE [33].

9. Conclusions

We have proposed an approach to detect infeasible paths during dynamic test data generation process. Our approach combines the use of empirical properties of infeasible paths with dynamic information collected during test data generation to detect the infeasibility of the path under consideration with high degree of precision. Given a path in a program, the program is initially executed with an arbitrarily chosen input. The execution of the program is monitored. If an undesired branch is taken due to a pair of e-correlated conditional statements, it is highly probable that the path is infeasible. If this is the case, the test data generation process immediately terminates; otherwise, the input is refined such that the desired branch is taken. This process is repeated until the input is found or the path is concluded as infeasible.

We have conducted a binomial test to validate Empirical Property 1. Other empirical properties can be proved from Empirical Property 1. The validation gives evidence that Empirical Property 1 holds for more than 90% of all the cases at 0.05% level of significance. We also conduct an experiment to compare the performance of the proposed approach with the infeasible path detection heuristics proposed by Bueno et al. [4]. The proposed approach results in 4.03% false positive cases, which leads to an average precision of 96.02% and a recall of 100% in all the cases. More important, the number of false positive cases produced by the proposed approach is 2.42 times lesser as compared with the Bueno approach. This suggests that the proposed infeasible path detection technique can be used in the current industry structural testing practice, which requires only 80–90% coverage. Future work will focus on using simple symbolic substitution or incorporate more empirical properties to handle the false positive cases. 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 the infeasibility of a path during the test data generation process.

The novelty of our approach lies in the use of empirical properties, which provide a simple yet efficient approach to detect the infeasibility of a path. 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.

References

[1] Numerical methods for constrained optimization, Academic Press, 1974.

[2] R. Bodik, R. Gupta, M.L. Soffa, Interprocedural conditional branch elimination, presented at Proceedings of the 1997 ACM SIGPLAM Conference on Programming Language Design and Implementation, PLDI, June 15–18 1997, Las Vegas, NV, USA, 1997.

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

[4] P.M.S. Bueno, M. Jino, Identification of potentially infeasible program paths by monitoring the search for test data, presented at Proceedings of ASE 2000 15th IEEE International Automated Software Engineering Conference, 11–15 September 2000, Grenoble, France, 2000.

[5] P.M.S. Bueno, M. Jino, Automatic test data generation for program paths using genetic algorithms, International Journal of Software Engineering and Knowledge Engineering 12 (2002) 691–709.

[6] L.A. Clarke, A system to generate test data and symbolically execute programs, IEEE Transactions on Software Engineering SE-2 (1976) 215–222.

[7] C. Csallner, Y. Smaragdakis, Check 'n' crash: combining static checking and testing, presented at 27th International Conference on Software Engineering, ICSE 2005, May 15–21 2005, Saint Louis, MO, United States, 2005.

[8] R. Ferguson, B. Korel, The chaining approach for software test data generation, ACM Transactions on Software Engineering and Methodology 5 (1996) 63–86.

[9] J. Ferrante, K.J. Ottenstein, J.D. Warren, The program dependence graph and its use in optimization, ACM Transactions on Programming Languages and Systems 9 (1987) 319–349.

[10] I. Forgacs, A. Bertolino, Feasible test path selection by principal slicing, presented at ESEC/FSE '97, 6th European Software Engineering Conference Held Jointly with the 5th ACM SIGSOFT Symposium on the Foundations of Software Engineering, 22–25 September 1997 Software Engineering Notes, Zurich, Switzerland, 1997.

[11] K.B. Gallagher, J.R. Lyle, Using program slicing in software maintenance, IEEE Transactions on Software Engineering 17 (1991) 751–761.

[12] B. George, L. Williams, A structured experiment of test-driven development, Information and Software Technology 46 (2004) 337–342.

[13] GFP, A Personal Financial Management System. Available from: <http://gfd.sourceforge.net/>.

[14] A. Goldberg, T.C. Wang, D. Zimmerman, Applications of feasible path analysis to program testing, presented at 1994 International Symposium on Software Testing and Analysis (ISSTA), 17–19 August 1994 SIGSOFT Software Engineering Notes, Seattle, WA, USA, 1994.

[15] GraphViz, Dotty. Available from: <http://www.graphviz.org/>.

[16] N. Gupta, A.P. Mathur, M.L. Soffa, Automated test data generation using an iterative relaxation method, Proceedings of the 1998 ACM SIGSOFT 6th International Symposium on the Foundations of Software Engineering, FSE-6, SIGSOFT-98, November 3–November 5 1998, pp. 231–244, 1998.

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

[18] D. Hedley, M.A. Hennell, The causes and effects of infeasible paths in computer programs, presented at Proceedings of the 8th International Conference on Software Engineering (Cat. No.85CH2139-4), 28–30 August 1985, London, UK, 1985.

[19] HTMLParser, A HTML Parser. Available from: <http://htmlparser.sourceforge.net/>.

[20] G. Jayaraman, V.P. Ranganath, J. Hatcliff, Kaveri: delivering the Indus Java program slicer to Eclipse, presented at Fundamental Approaches to Software Engineering. 8th International Conference, FASE 2005 Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2005. Proceedings, 4–8 April 2005, Edinburgh, UK, 2005.

[21] JHotDraw, A Java GUI framework. Available from: <http://www.jhotdraw.org/>.

[22] JLibSys, A Library Management System. Available from: <http://sourceforge.net/projects/jlibsystem/>.

[23] JMathLib, A library of mathematical functions. Available from: <http://mathlib.sourceforge.net/>.

[24] B. Korel, Automated software test data generation, IEEE Transactions on Software Engineering 16 (1990).

[25] B. Korel, Automated test data generation for programs with procedures, presented at Proceedings of International Symposium on Software Testing and Analysis (ISSTA'96), 8–10 January 1996, San Diego, CA, USA, 1996.

[26] N. Malevris, A path generation method for testing LCSAJs that restrains infeasible paths, Information and Software Technology 37 (1995) 435–441.

[27] N. Malevris, D.F. Yates, A. Veevers, Predictive metric for likely feasibility of program paths, Information and Software Technology 32 (1990) 115–118.

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

[29] C.C. Michael, G. McGraw, M.A. Schatz, Generating software test data by evolution, IEEE Transactions on Software Engineering 27 (2001) 1085–1110.

[30] R.P. Pargas, M.J. Harrold, R.R. Peck, Test-data generation using genetic algorithms, Software Testing Verification and Reliability 9 (1999) 263–282.

[31] PMD, A Java source code analyzer. Available from: <http://pmd.sourceforge.net/>.

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

[33] K. Sen, G. Agha, CUTE and jCUTE: concolic unit testing and explicit path model-checking tools, presented at 18th International Conference on Computer Aided Verification, CAV 2006, August 17–20 2006, Seattle, WA, United States, 2006.

[34] SOOT, A Java bytecode optimization framework. Available from: <http://www.sable.mcgill.ca/soot/>.

[35] Sourceforge, Open-source website. Available from: <http://sourceforge.net/>.

[36] P. Tonella, Evolutionary testing of classes, presented at ACM SIGSOFT International Symposium on Software Testing and

Analysis, ISSTA 2004, July 11–14 2004, Boston, MA, United States, 2004.

[37] W. Visser, C.S. Pasareanu, S. Khurshid, Test input generation with Java PathFinder, presented at ACM SIGSOFT International Symposium on Software Testing and Analysis – ISSTA 2004, 11–14 July 2004 Software Engineering Notes, Boston, MA, USA, 2004.

[38] A. Watkins, E.M. Hufnagel, Evolutionary test data generation: a comparison of fitness functions, Software – Practice and Experience 36 (2006) 95–116.

[39] T. Xie, D. Marinov, W. Schulte, D. Notkin, Symstra: a framework for generating object-oriented unit tests using symbolic execution, presented at Tools and Algorithms for the Construction and Analysis of Systems, 11th International Conference, TACAS 2005, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2005, Proceedings, 4–8 April 2005, Edinburgh, UK, 2005.

[40] D.F. Yates, N. Malevris, Reducing the effects of infeasible paths in branch testing, presented at ACM SIGSOFT '89 Third Symposium on Software Testing, Analysis and Verification (TAV3), 13–15 December 1989 SIGSOFT Software Engineering Notes, Key West, FL, USA, 1989.

[41] J. Zhang, X. Chen, X. Wang, Path-oriented test data generation using symbolic execution and constraint solving techniques, presented at Proceedings of the Second International Conference on Software Engineering and Formal Methods, 28–30 September 2004, Beijing, China, 2004.

[42] J. Zhang, X. Wang, A constraint solver and its application to path feasibility analysis, International Journal of Software Engineering and Knowledge Engineering 11 (2001) 139–156.

[43] R. Zhao, M.R. Lyu, Character string predicate based automatic software test data generation, IEEE 3rd International Conference on Quality Software (2003).