Path-Directed Source Test Case Generation and Prioritization in Metamorphic Testing

Chang-ai Sun[a,*], Baoli Liu[a], An Fu[a], Yiqiang Liu[a], Huai Liu[b]

[a]*School of Computer and Communication Engineering, University of Science and Technology Beijing, Beijing 100083, China*
[b]*Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn VIC 3122, Australia*

Abstract

Metamorphic testing is a technique that makes use of some necessary properties of the software under test, termed as metamorphic relations, to construct new test cases, namely follow-up test cases, based on some existing test cases, namely source test cases. Due to the ability of verifying testing results without the need of test oracles, it has been widely used in many application domains and detected lots of real-life faults. Numerous investigations have been conducted to further improve the effectiveness of metamorphic testing, most of which were focused on the identification and selection of "good" metamorphic relations. Recently, a few studies emerged on the research direction of how to generate and select source test cases that are effective in fault detection. In this paper, we propose a novel approach to generating source test cases based on their associated path constraints, which are obtained through symbolic execution. The path distance among test cases is leveraged to guide the prioritization of source test cases, which further improve the efficiency. A tool has been developed to automate the proposed approach as much as possible. Empirical studies have also been conducted to evaluate the fault-detection effectiveness of the approach. The results show that this approach enhances both the performance and automation of metamorphic testing. It also highlights interesting research directions for further improving metamorphic testing.

Keywords: Metamorphic testing, source test case, symbolic execution, path constraint

1. Introduction

Software testing, a major approach to software quality assurance, is mainly targeted at demonstrating "the presence of bugs" [1, p.16]. A mainstream way to implement testing is to dynamically execute the software under test (SUT): Test cases are first generated according to some objectives, such as achieving some degree of code coverage and detecting certain types of fault. After the execution of a test case, the actual result will be verified against the expected output, normally through a systematic mechanism, termed as *test oracle*.

The basic processes of test case generation and test result verification are concerned with two fundamental problems of software testing, namely the *reliable test set problem* and the *oracle problem* [2], respectively. The former problem basically refers to the infeasibility of exhaustive testing — It is almost impossible to exhaustively execute all possible test cases even for a simple program, so a subset of test cases should be deliberately selected to provide a reliable coverage of as many functionalities of SUT as possible. A large number of techniques have been developed to generate test cases from different perspectives [3, 4, 5]. However, most of them have assumed, at least implicitly, the presence of a test oracle. Their fault-detection capabilities will be influenced by the oracle problem; in other words, when either there is no test oracle or the oracle is too expensive to apply for verifying the correctness of test results, the applicabilities of many test case generation techniques are significantly limited.

Among all testing techniques, *metamorphic testing* (MT) [6, 7] is a unique one that can not only address the oracle problem but also provide a new way for generating test cases. The core element of MT is a set of so-called *metamorphic relations* (MRs), which are the necessary properties of SUT presented in the form of relationships among multiple program inputs and their expected outputs. In the aspect of test case generation, MRs can be used to transform some existing test cases (termed as *source test cases* in MT) into new test cases (termed as *follow-up test cases*). For the test result verification, instead of using an oracle for each individual test case, the test results from multiple test cases are checked against the corresponding MRs.

Despite the simplicity in concept, MT has been very successful in revealing many real-life bugs in a wide variety of application domains. For example, simple equality MRs have led to the detection of thousands of erroneous behaviors in some top-ranked autonomous driving models — Fatally wrong decisions could be made given a minor change in the weather or road conditions [8, 9]. Xie et al. [10] also used MT to validate traditional machine learning classifiers and found that some of their behaviours were not consistent with users' expectations. In addition, MT have revealed hundreds of faults in some widely used compilers [11]. The high fault-detection capability of MT is not only due to its ability to address the oracle problem, but also because it can generate test cases that are complimentary to those created by traditional testing techniques [6].

In recent years, the identification of MRs, the core element of MT, has received considerable attention in the community of MT [12, 13, 14, 15, 16, 17, 18, 19]. Some researchers investigated the attributes of "good" MRs, that is, what MRs have high potentials in detecting software faults. Liu et al. [18], for example, justified that a small number of "diverse" MRs are sufficient by themselves to detect the most faults that can be revealed using a test oracle. Other studies were aimed at the systematic ways for identifying MRs. For instance, the METRIC approach [12] was proposed to construct MRs based on the concepts of category and choice in the category-partition method [20]. Recently, it was extended to the METRIC+ approach [13] through the inclusion of output-related categories and choices and the introduction of more systematic mechanism for deciding appropriate relations.

As another critical component, source test cases also play an important role in MT. Since the follow-up test cases (that is, the new test cases created by MT) are constructed based on source test cases and MRs, the quality of source test cases is also a driving factor for the performance of MT. In the majority of previous studies of MT, source test cases were normally generated using random testing, which could provide a good benchmark as the performance's lower bound in empirical studies. By nature, some more systematic techniques could be applied in the source test case generation and selection to improve MT's performance. Barus et al. [21], for example, utilized the so-called adaptive random testing (ART) [3] to improve the diversity of source test cases in MT, which, in turn, enhanced the fault-detection effectiveness of MT. In addition, Alatawi et al. [22] generated source test cases based on the dynamic symbolic execution [4] and showed promising performance improvement. Some researchers used the test cases of MT in an iterative way [23, 24]: Some follow-up test cases could be re-used the source test cases for the next round of testing. Dong et al. [25] proposed a technique to generate source test cases based on genetic algorithm and program path analysis.

In line with the research of source test case generation, this paper attempts to investigate how to maximize the diversity of source test cases based on their path constraints, which are obtained through the symbolic execution [26]. New techniques are proposed for improving the overall performance of MT. A comprehensive framework is developed to facilitate the automatic implementation of MT. The paper makes the following four major contributions:

- The symbolic execution is applied to construct path constraints, which, in turn, guide the generation of source test cases of MT to guarantee (i) that different program paths are thoroughly covered, and (ii) that test cases are diversified in terms of the execution behaviors triggered by them.
- A path distance is defined to guide the prioritization of source test cases for an efficient execution of MT.
- A prototype tool is developed to integrate all new techniques into existing MT approach and thus to enable automated testing.
- Empirical studies are conducted based on real-world programs and demonstrate the performance improvement brought by the new techniques over traditional MT.

The rest of the paper is organized as follows. Section 2 introduces the background information for this study. The new techniques and the prototype tool are described in Section 3. Section 4 presents the design and settings of our experiments, the results of which are given in Section 5. The studies related to our work are discussed in Section 6. Finally, Section 7 summarizes the paper with pointing out future work.

2. Background

2.1. Metamorphic Testing

As discussed above, MT supports test case generation process and provides a test result verification mechanism, both on the basis of MRs. The basic procedure of MT is as follows:

1. Select one MR, which represents a necessary property of SUT in the form of the relationship among multiple inputs and their expected outputs.
2. Generate source test case(s) using some existing testing techniques.
3. Construct follow-up test case(s) by applying the MR to transform source test case(s).
4. Execute source and follow-up test cases, which collectively are termed as *metamorphic test group*.
5. Verify the execution results of the metamorphic test group against the MR. If the MR is violated, the SUT is considered faulty; otherwise, the SUT passes the testing of the corresponding metamorphic test group.

Note that the basic concept of MR requires that each metamorphic test group contains at least one source test case and at least one follow-up test case (as shown in the above Steps 2 and 3). As an example to illustrate, suppose a program P calculates the sine value for an angle as the input. One possible MR for P is: Given $a = b + c$, we should have the relation $\sin(a) = \sin(b + c) = \sin(b)\cos(c) + \cos(b)\sin(c) = \sin(b)\sin(90° - c) + \sin(90° - b)\sin(c)$. When implementing MT on P based on this MR, we generate a source test case x, and then construct two follow-up test cases y and z, where $x = y + z$. There will be five executions of SUT, $P(x)$, $P(y)$, $P(z)$, $P(90° - y)$, and $P(90° - z)$, and we need to check whether the relation $P(x) = P(y)P(90° - z) + P(90° - y)P(z)$ holds. Any violation of the relation will imply the detection of a fault.

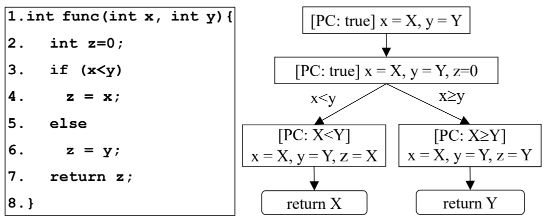
For the "existing testing techniques" in Step 2, the majority of previous studies have randomly generated source test cases [7]. Chen et al. [27] investigated the usage of special values as the source test cases for MT. Some studies [28] used the existing "classical" real-world inputs as the source test cases. Recently, more advanced techniques have been applied to generate "good" source test cases [21, 22, 25]. The present work is focused on the source test case generation based on the technique of symbolic execution, which is introduced in the next section.

2.2. Symbolic Execution

The core notion of symbolic execution [26] is to use the symbolic values to represent the program input parameters, and correspondingly to utilize symbolic expressions to denote the variables in program execution and the output results. The program analysis and validation will be implemented through the simulation of program executions. There are three main components in each step of the symbolic execution:

1. *Path condition*: Also termed as *path constraint*, a path condition is represented by a series of branch conditions. It denotes a certain path for program execution, which, in turn, can refer to all inputs that satisfy the corresponding branch conditions.
2. *Program counter*: It defines the statement to be executed next.
3. *Symbolic values for program variables*: Any variable in the program will be represented by a symbolic expression.

After combining the state of each execution step, we can obtain a symbolic execution tree, which gives all the execution paths. In the execution tree, each node refers to a program state, and the state transition is denoted by the edge. Figure 1 gives a simple example of a program and its corresponding execution tree. At the beginning, the path condition is "True". During the program execution, the variables x and y are replaced by the symbolic values X and Y, respectively. When the execution reaches the conditional statement (line 3), there will be two paths, representing the scenarios of "condition is satisfied" and "condition is not satisfied", respectively. Correspondingly, path conditions will become $X < Y$ and $X \geq Y$, which, in turn, lead to two different symbolic outputs $z = X$ and $z = Y$, respectively. In summary, we obtain two execution paths and their conditions.



```
int func(int x, int y){
    int z=0;
    if (x<y)
      z = x;
    else
      z = y;
    return z;
}
```

[PC: true] x = X, y = Y
[PC: true] x = X, y = Y, z=0
x<y
x≥y
[PC: X<Y] x = X, y = Y, z = X
[PC: X≥Y] x = X, y = Y, z = Y
return X
return Y

Figure 1: A simple program and its symbolic execution tree

Since the invention of symbolic execution, different techniques have been proposed to implement it under various scenarios. One main methodology is the so-called concolic execution [22, 29], the mixture of symbolic and concrete execution. For example, Godefroid et al. [4] developed a method, namely directed automated random testing (DART), which, starting from an input with randomly generated concrete value, executes the SUT both symbolically and concretely. The concrete execution will direct the symbolic execution on the same path; at the same time, path constraints will be extracted once a branch is reached. Different paths can be explored by negating certain path constraints. DART, also termed as dynamic symbolic execution (DSE), has been widely used in a variety of application domains, including the generation of source test cases in MT [22].

Tools have also been developed for implementing symbolic execution in different paradigms. Symbolic PathFinder (SPF) [30, 31], for example, combines the symbolic execution with Java PathFinder (JPF) [32], a model checker for Java programs. Working on the virtual machine supplied by JPF, SPF can systematically search different symbolic execution paths. SPF also makes use of some strategies provided in JPF, such as depth-first and breadth-first, to conduct the search. For the path constraints extracted during symbolic execution, their conditions are obtained through the constraint solver. Test cases can be created to satisfy these path conditions and thus represent different execution paths. SPF has the following advantages: (1) It does not require seeded inputs to drive symbolic execution [4, 31]; (2) It does not involve code instrumentation, thus ensuring high efficiency and applicability [4, 31]; (3) It is powerful in handling complex path constraints, supporting iterative constraint solving for hard-to-solve expressions. Inspired by the above advantages, we will use SPF to implement symbolic execution for source test case generation [31, 33].

2.3. Test Case Prioritization

Test case prioritization is a major activity in regression testing [34]. Basically speaking, regression testing re-runs some existing test cases, namely regression test cases, after changes are made to SUT. Its main aim is to verify whether the existing functionalities are affected by the changes — The functions that have passed the regression test cases should pass again after the changes are made. Various test cases may have different effectiveness in the regression testing, so it is advisable to schedule their execution order (in other words, prioritizing them) for maximizing the testing efficiency.

One mainstream approach to test case prioritization is based on the code coverage [35]. Some prioritization techniques order test cases based on their "total" coverage of certain code elements, such as the absolute number of statements or branches covered by them. Other techniques make use of the "additional" coverage, that is, how many statements/branches that can be covered by new test cases but not by the already selected ones.

Although originally proposed in the context of regression testing, the prioritization of test cases can be applied into other testing activities — It is always preferable to run the "good" test cases early to achieve some goals as quickly as possible, such as detecting faults and realizing certain coverage. In this study, we propose prioritizing the execution order of source test cases and thus the whole metamorphic test group for further improving the performance of MT.

3. Approach and Tool

In this section, we first discuss the motivation of our work. Then, the approach is described, including both techniques for source test case generation and prioritization, which are illustrated by an example. Finally, we present the tool we have developed to implement the approach.

3.1. Motivation

Intuitively speaking, to guarantee a high potential of detecting a wide variety of faults, test cases should enable a testing method, like MT, to trigger as different execution behaviors as possible, including those "hard-to-reach" execution paths/statements, which sometimes even deserve substantial efforts. The symbolic execution technique [26] provides a direct and effective solution to this issue. It has attracted increasing attentions due to its capability of analyzing program behaviors. The technique represents the program inputs using symbolic values (rather than concrete values) to explore possible program execution paths and collect constraints of program branches covered by a path. Apparently, symbolic execution can be used to analyze the constraints of hard-to-reach statements. By solving the constraints, test cases that execute these program paths are generated. Motivated by its capability of thoroughly covering execution paths, we propose the application of symbolic execution into the generation of source test cases for MT.

Any software development activity is constrained by resources. For software testing, its main goal is to detect as many faults as possible with limited budget and time. In other words, testing must be implemented in a cost-effective way. Normally, test cases are executed in a certain order. In all previous studies of MT, unfortunately, the execution order of test cases was either random or arbitrary; that is, no systematic prioritization has been applied for MT's test cases. In this study, we propose a prioritization technique that schedules the execution order of source test cases such that those with higher potentials of revealing faults are ranked with higher priorities and hence executed earlier. More specifically, the technique determines the priorities of source test cases based on their contributions to the coverage of program paths/statements. Source test cases that help achieve the highest statement coverage are executed first. We particularly adopted the path-directed prioritization for source test cases due to the following two reasons: (1) Basically, the execution paths of test cases reflect certain functionalities of SUT, so the more different paths are exercised by test cases, the more functionalities will be covered by them and thus higher likelihood they have in revealing potential faults; (2) Since it is relatively complicated to control the paths to be covered by follow-up test cases (the construction of which relies on both source test cases and MRs), we design simple prioritization techniques for source test cases in this study.

The proposed source test case generation and prioritization techniques compose our approach, as elaborated in the following section.

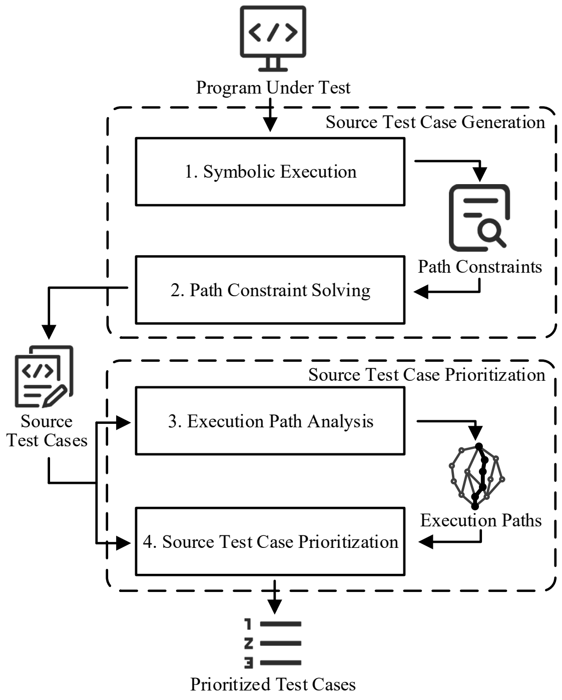

Program Under Test
Source Test Case Generation
1. Symbolic Execution
Path Constraints
2. Path Constraint Solving
Source Test Cases
Source Test Case Prioritization
3. Execution Path Analysis
Execution Paths
4. Source Test Case Prioritization
Prioritized Test Cases

Figure 2: Approach overview

3.2. Our Approach

Figure 2 outlines the overview of our approach, which consists of the following steps:

(1) *Symbolic Execution*: SUT is symbolically executed to explore its possible execution paths and acquire the corresponding path constraints.

(2) *Path Constraint Solving*: Constraint solver is employed to generate source test cases that satisfy the constraints of each possible execution path.

(3) *Execution Path Analysis*: For each generated source test case, the sequence of executed statements is analyzed to form the execution path.

(4) *Source Test Case Prioritization*: Source test cases are prioritized based on the *path distance* (to be defined in Section 3.2.2 below) among test cases.

The first and second steps constitute the source test case generation phase (explained in Section 3.2.1), and the third and fourth steps constitute the source test case prioritization phase (Section 3.2.2).

3.2.1. Path-directed source test case generation

To perform symbolic execution on SUT, we select Symbolic PathFinder (SPF) [30, 36] as our symbolic execution engine. SPF is an extension of Java PathFinder [37], which is a well-known model checker for Java Programs. SPF provides symbolic execution for bytecode of Java programs. In addition, SPF supports the symbolization of multiple types of input parameters, which makes it applicable to most Java programs. SPF takes the bytecode of SUT and a configuration file as input, and outputs the constraints of possible execution paths. The configuration file specifies which method to execute symbolically and which method arguments are symbolized. The tool constructs and explores the symbolic execution tree of SUT, in which each path from the root to a leaf represents a possible execution path of SUT. Note that symbolic execution suffers from some problems, such as the path explosion problem, which can happen particularly when loops involve symbolic values. In this study, we follow the traditional solution of limiting the number of iterations (up to 30) to address the problem.

Choco-solver [38, 39] is used as our constraint solving tool. Choco-solver is an open-source Java library for solving the constraint satisfaction problem (CSP). The tool allows the user to model a CSP by stating a set of variables with their constraints that must be satisfied. Then the tool leverages search-based algorithms to find values of variables that satisfy the stated constraints. In this study, the path constraints obtained from SPF serve as inputs of Choco-solver. Accordingly, the solver outputs suitable values of variables that make the constraints true. These values together constitute the source test case that executes a certain program path.

To illustrate the proposed source test case generation technique, let us look at an example method `MyMethod`, as below.

```
public int myMethod(int x, int y){
    int z = x + y;
    if (z == 0){
        if (y > 0){
            z = y − x;
        }else{
            z = x − y;
        }
    }else{
        if (x > 0){
            z = z − x;
        }else{
            z = z + x;
        }
    }
    return z;
}
```

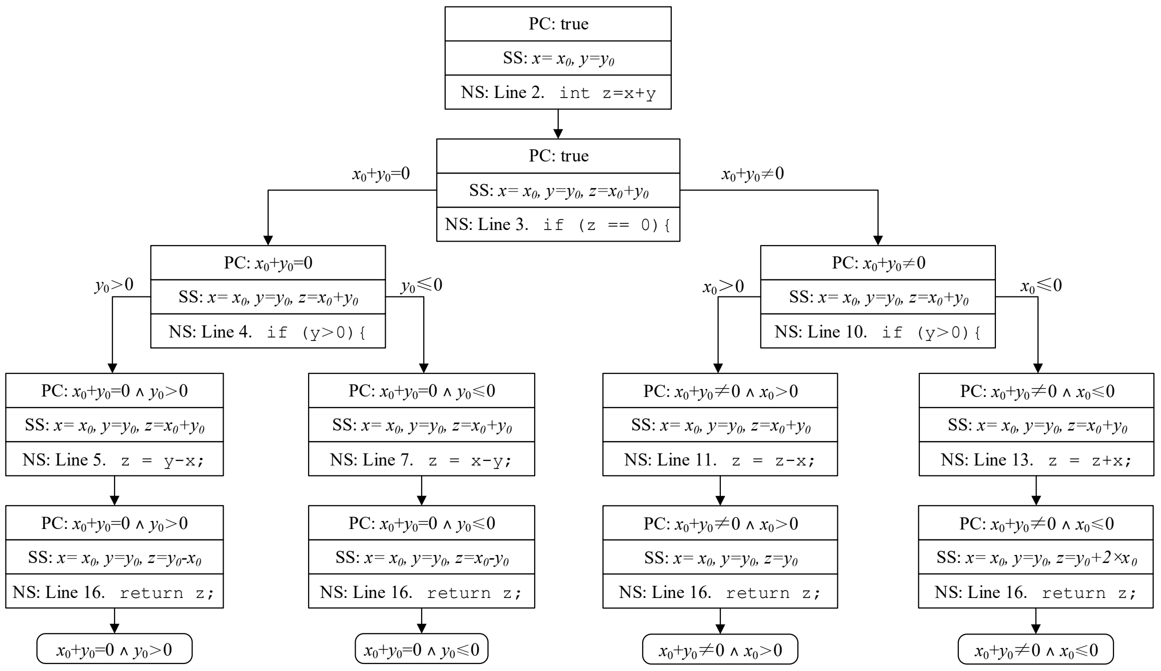
In *Step 1* of our approach (Figure 2), `MyMethod` is symbolically executed using SPF. The result of symbolic execution is represented as the symbolic execution tree shown in Figure 3. The path constraint of each state in the tree specifies the conditions that input parameters must satisfy to trigger the execution of that state. Accordingly, the path constraint of a leaf node in the tree specifies the conditions for executing a program path. The path constraints[1] of leaf nodes are summarized in the "Path Constraint" column of Table 1.

In *Step 2* of Figure 2, the path constraints are solved by Choco-solver to generate values for input parameters that makes the constraints true. The generated test cases are shown in the "Generated Source TC" column of Table 1.

It should be noted that for the first two paths in Table 1, it is hard to generate source test cases that execute these paths by selecting random values or special values for x and y, respectively. The reason is that these paths have strict constraints (i.e., (y_0_SYMINT+x_0_SYMINT)==CONST_0). Apparently, under circumstance where such constraint is unknown, it is not easy to select random values or special values for x and y that satisfy this constraint, and thereby exercising statements controlled by this constraint could be difficult. Our approach can obtain such constraint and generate source test cases that satisfy the constraint, which makes it possible to effectively cover the hard-to-reach program statements when conducting MT.

Note that some inherent limitations of constraint solving may hinder the effectiveness of symbolic execution [26], and thus affect the performance of our source test case generation approach. For example, the constraint solver may fail to return a viable solution when path constraints are too complicated; and those constraints that involve non-linear arithmetic are normally undecidable [40]. Having said

[1]Note that "y_0_SYMINT" represents a symbolized integer variable y_0, "x_0_SYMINT" represents a symbolized integer variable x_0, and "CONST_0" represents a constant whose value is 0.



PC: true
SS: $x= x_0, y=y_0$
NS: Line 2. int z=x+y
PC: true
SS: $x= x_0, y=y_0, z=x_0+y_0$
NS: Line 3. if (z == 0){
$x_0+y_0=0$
$x_0+y_0\neq0$
PC: $x_0+y_0=0$
SS: $x= x_0, y=y_0, z=x_0+y_0$
NS: Line 4. if (y>0){
$y_0>0$
$y_0\leqslant0$
PC: $x_0+y_0\neq0$
SS: $x= x_0, y=y_0, z=x_0+y_0$
NS: Line 10. if (y>0){
$x_0>0$
$x_0\leqslant0$
PC: $x_0+y_0=0 \wedge y_0>0$
SS: $x= x_0, y=y_0, z=x_0+y_0$
NS: Line 5. z = y-x;
PC: $x_0+y_0=0 \wedge y_0\leqslant0$
SS: $x= x_0, y=y_0, z=x_0+y_0$
NS: Line 7. z = x-y;
PC: $x_0+y_0\neq0 \wedge x_0>0$
SS: $x= x_0, y=y_0, z=x_0+y_0$
NS: Line 11. z = z-x;
PC: $x_0+y_0\neq0 \wedge x_0\leqslant0$
SS: $x= x_0, y=y_0, z=x_0+y_0$
NS: Line 13. z = z+x;
PC: $x_0+y_0=0 \wedge y_0>0$
SS: $x= x_0, y=y_0, z=y_0-x_0$
NS: Line 16. return z;
PC: $x_0+y_0=0 \wedge y_0\leqslant0$
SS: $x= x_0, y=y_0, z=x_0-y_0$
NS: Line 16. return z;
PC: $x_0+y_0\neq0 \wedge x_0>0$
SS: $x= x_0, y=y_0, z=y_0$
NS: Line 16. return z;
PC: $x_0+y_0\neq0 \wedge x_0\leqslant0$
SS: $x= x_0, y=y_0, z=y_0+2\times x_0$
NS: Line 16. return z;
$x_0+y_0=0 \wedge y_0>0$
$x_0+y_0=0 \wedge y_0\leqslant0$
$x_0+y_0\neq0 \wedge x_0>0$
$x_0+y_0\neq0 \wedge x_0\leqslant0$

Figure 3: Symbolic execution tree of `MyMethod`

Table 1: A summary of path constraints of `MyMethod` and generated source test cases.

Path ID	Path Constraint	Generated Source TC
1	y_0_SYMINT>CONST_0 && (y_0_SYMINT+x_0_SYMINT)==CONST_0	$t_1 : x_0 = -5, y_0 = 5$
2	y_0_SYMINT≤CONST_0 && (y_0_SYMINT+x_0_SYMINT)==CONST_0	$t_2 : x_0 = 2, y_0 = -2$
3	x_0_SYMINT>CONST_0 && (y_0_SYMINT+x_0_SYMINT)! =CONST_0	$t_3 : x_0 = 3, y_0 = 2$
4	x_0_SYMINT≤CONST_0 && (y_0_SYMINT+x_0_SYMINT)! =CONST_0	$t_4 : x_0 = -4, y_0 = -1$

that, Choco-solver used in our study had worked very well in the sense of ensuring a good performance of our technique.

3.2.2. Path-directed source test case prioritization

Before the introduction of our test case prioritization technique, we first give the following definitions as the theoretical foundation.

Definition 1 (*path*)**.** *A path is defined as a sequence of statements that a given source test case sequentially (or continuously) executes during a program run.*

Given a program P which consists of a set of statements $P = \{s_1, s_2, \ldots, s_n\}$. Consider a source test case t that executes the following statements one by one until the program exits: s_1, s_3, s_5, s_3, s_7. The path of t is the sequence composed of the aforementioned statements, denoted as $path(P, t) =< s_1, s_3, s_5, s_3, s_7 >$.

Definition 2 (*statement set*)**.** *A statement set is defined as a set composed of elements in the statement sequence of a path.*

Recall t with its $path(P, t) =< s_1, s_3, s_5, s_3, s_7 >$. The corresponding statement set is denoted as $set(P, t) = \{s_1, s_3, s_5, s_7\}$. Apparently, a statement set includes the statements covered by a given source test case. Similarly, a statement set of a given set of test cases $TS = \{t_1, t_2, \ldots, t_n\}$ is denoted as $set(P, TS)$, which includes all the statements covered by test cases in TS, that is,

$set(P, t_1) \cup set(P, t_2) \cup \cdots \cup set(P, t_n)$.

Definition 3 (*path length*). *The length of a path is defined as the size of statement set of a path.*

Recall t with its $path(P, t)$ and $set(P, t)$. The length of $path(P, t)$ is $|set(P, t)| = 4$.

On the basis of the aforementioned definitions, we define the path distance between a set of test cases and a single test case as follows.

Definition 4 (*path distance*). *The path distance between a set of test cases TS and a single test case t is defined as the number of statements covered by t but not covered by TS.*

Recall t with its statement set $\{s_1, s_3, s_5, s_7\}$. Given a test case set TS with statement set $set(P, TS) = \{s_1, s_2, s_6, s_7, s_8\}$. The path distance between t and TS is calculated as $dist(t, TS) = |set(P, t) \setminus set(P, TS)| = |\{s \mid s \in set(P, t) \wedge s \notin set(P, TS)\}| = 2$.

Based on the aforementioned definitions, we propose an algorithm to prioritize source test cases. The input of our algorithm is a collection of key-value pairs $KV = \{< t_1, set(P, t_1) >, \ldots, < t_n, set(P, t_n) >\}$. For each of the key-value pair in KV, the key represents a source test case and the value represents the corresponding statement set. The output of our algorithm is a list of prioritized source test cases L_{pri} where the higher the ranking in the list, the higher the execution priority.

Algorithm 1: Source test case prioritization based on path distance

```
   Input: $KV = \{< t_1, set(P, t_1) >, < t_2, set(P, t_2) >, \ldots, < t_n, set(P, t_n) >\}$
   Output: $L_{pri}$
1  Initialize $L_{pri}$    EmptyList(), $S_{pri} \leftarrow \emptyset$;
2  while $KV \neq \emptyset$ do
3      Initialize $d = 0, max = 0, t_{max}$    $NULL$;
4      for each key-value pair $< t_i, set(P, t_i) >$ in $KV$ do
5          $d = dist(t_i, S_{pri}) = | set(P, t_i) \setminus set(P, S_{pri}) |$;
6          if $d > max$ then
7              $max = d$;
8              $t_{max}$    $t_i$;
9          end
10     end
11     $S_{pri}$    $S_{pri} \cup \{t_{max}\}$;
12     $L_{pri}$    Append($t_{max}$);
13     $KV$    $KV \setminus \{< t_{max}, set(P, t_{max}) >\}$;
14 end
15 return $L_{pri}$
```

In Algorithm 1, the ranking list (L_{pri}) and the set of prioritized source test cases (S_{pri}) are first initialized as empty (Line 1). For each key-value pair in KV, the algorithm searches for the test case that has the longest path distance from S_{pri} (Lines 3-10). Then, the source test case (t_{max}) in the resulting key-value pair is added to S_{pri} and appended to L_{pri} (Lines 11-12). In addition, $< t_{max}, set(P, t_{max}) >$ is removed from KV before the next iteration starts. The steps in lines 3-13 are repeated until KV is empty. Apparently, the list of prioritized source test cases is incrementally updated with the test case that is farthest from the already prioritized ones.

Intuitively speaking, the basic principle of our prioritization technique is to rank the source test cases based on the extent to which they can increase the coverage of statements/paths. Theoretically speaking, since each generated source test case corresponds to a path in the symbolic execution tree of SUT, the execution path of a given source test case can be obtained based on the mapping relationship between the symbolic execution tree and the source code. In practice, we obtain the execution path of a source test case by monitoring the output of special statements that are instrumented in the beginning each of the branches and loops of the SUT.

Let us revisit the example given in Section 3.2.1 to illustrate the prioritization technique. After *Step 3* in our approach (Figure 2), we get the execution paths of generated source test case, as shown in Table 2. Then, in *Step 4*, the source test cases are prioritized based on their execution paths. The source test cases selected in each iteration of our algorithm are shown in Table 3. The final outcome of the source test case prioritization is $< t_4, t_2, t_1, t_3 >$.

3.3. Supporting Tool

A tool has been developed to automate our approach as much as possible. The tool was also integrated with an existing tool, namely MT4WS [41], which supports MT for Web services. We particularly extended the "Test Case Generator" component of MT4WS to support our approach from two perspectives: (1) the integration of the SPF engine and Choco-solver into the component to support generation of source test cases, and (2) the implementation of Algorithm 1 in the component to support the prioritization of source test cases. More specifically, the supporting tool aids the following tasks.

1) *Configuration of Symbolic Execution*: This

Table 2: The execution paths of generated source test cases

Path ID	Source Test Case	Execution Path
1	$t_1 : x_0 = -5, y_0 = 5$	Line 1 → Line 2 → Line 3 → Line 4 → Line 5 → Line 16
2	$t_2 : x_0 = 2, y_0 = -2$	Line 1 → Line 2 → Line 3 → Line 4 → Line 6 → Line 7 → Line 16
3	$t_3 : x_0 = 3, y_0 = 2$	Line 1 → Line 2 → Line 3 → Line 9 → Line 10 → Line 11 → Line 16
4	$t_4 : x_0 = -4, y_0 = -1$	Line 1 → Line 2 → Line 3 → Line 9 → Line 10 → Line 12 → Line 13 → Line 16

Table 3: Details of each round of iteration of prioritization algorithm

Round of iteration	Remaining Source TCs	Elements in S_{pri}	$dist(t_i, S_{pri})$	t_{max}	L_{pri} after iteration
1	t_1 t_2 t_3 t_4	$\emptyset$	6 7 7 8	t_4	$< t_4 >$
2	t_1 t_2 t_3	t_4	2 3 1	t_2	$< t_4, t_2 >$
3	t_1 t_3	t_4, t_2	1 1	t_1	$< t_4, t_2, t_1 >$
4	t_3	t_4, t_2, t_1	1	t_3	$< t_4, t_2, t_1, t_3 >$

task allows the user to specify key settings of symbolic execution and outputs a configuration file for SPF. The key settings include the name of SUT, the method of interest to perform symbolic execution, and the method parameters that need to be symbolized. In addition, the user can specify the decision procedure of symbolic execution, the listener to print information about symbolic run, the search strategy and search depth of symbolic execution tree.

2) *Generation of Path Constraints*: SPF is employed to perform symbolic execution on the target method and outputs path constraints of SUT.

3) *Generation of Source Test Cases*: Choco-solver is used to solve the path constraints and generate values of variables that satisfy the path constraints. Then, source test cases are constructed using the generated values.

4) *Prioritization of Source Test Cases*: The priority of source test cases is ranked using our Algorithm 1. The outcome of this task is a list of source test cases in which higher ranking priority represents earlier execution.

4. Experimental Studies

Empirical studies were conducted to evaluate the performance of our approach, including both source test case generation and prioritization. This section presents the settings of experimental studies.

4.1. Research Questions

The experiments attempt to answer the following research questions:

RQ1 How effective is MT in fault detection using the source test cases generated by our approach?

RQ2 To what extent does our approach outperform existing source test case generation techniques?

RQ3 Can the proposed technique reveal program faults faster than baseline techniques with random execution order?

RQ4 What is the overhead of our approach for generating source test cases?

4.2. Object Programs

We selected seven objects programs representing different application domains from various sources. They are:

- Airlines Baggage Billing Service (BAGGAGE) [42] enables passengers to calculate their own baggage fees using the baggage charge scheme of Air China. BAGGAGE performs the calculation based on several factors, including the relevant flight, aircraft cabin, flight region, baggage weight, airfare, and eligibility for student discount.

- Phone Bill Calculation (PHONE) [42] is a mobile phone charge calculation system used by China Unicom. It computes a user's monthly phone charge based on the communication time, data usage, and mobile phone tariffs.

- Parking Fee Calculation (PARKING) [12] provides the calculation service of a vehicle's parking fee for a driver. It accepts the parking details including the vehicle type, day of the week, discount coupon, and hours of parking, and then calculates the parking fee according to the predefined hourly rates.

- Money Transfer Charging (CHAGRE)[2] simulates Alipay's money transfer operation from one account to another, and calculates the fee for money transfer. The program accepts the following information to make a transaction and charge for transfer fee: (1) the ID, balance, remaining amount of fee-free transfer of the account sending the money, (2) the ID and holder name of the account receiving the money, (3) amount of transfer, and (4) way of settlement.

- NumberUtil (NUMBER) [43] is a data type conversion program in the Apache Common Lang library for Java programming language. The program converts a number of `String` type into a number of `Number` type based on a series of type conversion rules. The `createNumber` method provides the primary conversion function of the program, which accepts a number of string form and outputs that number of Number type.

- Get Tax (TAX)[3] enables a customer to calculate the total amount of sale taxes paid for the purchased goods. TAX accepts the list of goods with amount, price, and rate of tax, and outputs the total amount of taxes.

- MathUtil (MATH) [43] is a utility class in the Apache Common Math library for Java programming language. It provides miscellaneous utility functions that address common mathematical problems such as the calculation of the greatest common divisor of two numbers, the normalization of an angle, and the distance calculation between two points.

All object programs are written in Java. The size of each object, measured by lines of code (LOC), is given in the second column of Table 4.

4.3. MR Identification

For programs BAGGAGE and PHONE, we reused existing MRs that were identified based on the μMT approach [42]. We examined the validity of the original MRs, and found that some MRs actually represent the same necessary properties of SUT, only with different forms. After merging these MRs, we finally obtained 18 MRs for each of the BAGGAGE and PHONE programs, as given in the third column of Table 4.

We also followed the μMT approach [42] to identify MRs for programs PARKING, CHARGE, NUMBER, TAX, and MATH. We defined their data mutation operators (DMOs) and mapping rules for output relations based on their specifications. By manually composing the DOMs and mapping rules, we finally derived 5, 20, 13, 24, and 3 MRs for PARKING, CHARGE, NUMBER, TAX, and MATH, respectively. Note that we also manually checked their validity to ensure that they correctly expressed the necessary properties of SUT.

Details of the identified MRs are available at https://github.com/PaDMT-USTB.

4.4. Faulty Versions of Object Programs

Our experiments used both artificially seeded faults (or mutants) and real-life faults. For programs BAGGAGE, PHONE, PARKING, CHARGE, TAX, and MATH, we applied mutation analysis [44] to generate mutants based on the MuJava tool [45]. Each mutant was generated by applying a singe syntactic change, namely the mutation operator, to a certain statement in the object program. It is well known that there exist some "equivalent mutants", which always have the same execution behaviors as the base program. To identify these equivalent mutants, we investigated the mutants that were not killed by all the test cases generated in this study, and manually checked

[2]https://github.com/PaDMT-USTB
[3]https://github.com/elainechan/sales-tax-calculator

Table 4: Basic information of object programs

Object program	LOC	# MRs	# Faulty versions	Source of faulty versions
BAGGAGE	101	18	56	MuJava with mutation operators AOIS, AORB, COI, LOI, ROR, AOIU
PHONE	113	18	112	MuJava with mutation operators AOIS, AOIU, AORB, COI, LOI, ROR
PARKING	266	5	754	MuJava with mutation operators AODU, AOIS, AORB, COD, COI, COR, LOI, ROR
CHARGE	1008	20	541	MuJava with mutation operators AODU, AOIS, AORB, AOIU, CDL, COD, COI, COR, LOI, ODL, ROR, SDL, VDL
NUMBER	1438	13	8	Real Bugs from Apache Common Lang in Defects4J Bug IDs: 1, 3, 5, 16, 27, 36, 44, 58
TAX	2150	24	1565	MuJava with mutation operators AODU, AOIS, AORB, AOIU, CDL, COR, COI, LOI, ODL, ROR
MATH	2002	3	435	MuJava with mutation operators AOIS, AORB, AOIU, CDL, COI, LOI, ODL, ROR, SDL, VDL

whether they are semantically equivalent to the base program. Since the number of candidate mutants was relatively small, it is feasible to do the manual checking. Finally, we generated 56, 112, 754, 541, 1565, and 435 non-equivalent mutants for BAGGAGE, PHONE, PARKING, CHARGE, TAX, and MATH, respectively.

For program NUMBER, a collection of real-world faults have been provided in an open-source project [43]. All these faults are located in the `NumberUtil` class. As a result, 8 faults (whose bug IDs are 1, 3, 5, 16, 27, 36, 44, and 58) were included in our experiments.

The fourth and fifth columns of Table 4 summarize the basic information of these faulty versions.

4.5. Generation of Test Cases

In our experiments, we first employed our approach to generate source test cases for all seven object programs. For each object program, we exhausted all possible execution paths obtained by SPF and attempted to solve their corresponding path constraints. After excluding the unsolvable path constraints, we generated 34, 32, 144, 25, 52, 18, and 68 source test cases for BAGGAGE, PHONE, PARKING, CHARGE, NUMBER, TAX, and MATH, respectively.

Note that some MRs may not be applicable to all source test cases. Therefore, before generating following-up test cases, we first determined a subset of applicable MRs for a certain source test case. Then, the follow-up test cases can be constructed based on the selected MRs.

4.6. Variables and Measurements

4.6.1. Independent variable

The independent variable in our experiments is related to the techniques under study. A natural choice is our approach based on ***Pa***th-***D***irected source test case generation and prioritization for ***MT*** (abbreviated as *PaDMT* hereafter).

Three baseline techniques were selected for comparison in RQ2 and RQ3. They are:

- RT (random testing based method): For each input parameter of an object program, we randomly generated a value within the valid value range of parameter. The generated values for input parameters together constituted a test case of an object program.
- ART (Adaptive random testing based method): We followed the previous study [21] to use ART to generate test cases – In addition to the random generation, test cases were further evenly spread across the whole input domain.
- DSE (Dynamic symbolic execution based method): The original work [22] applied DSE into C# programs. In this study, we adapted the method to Java programs with the support of SPF. DSE requires some initial test cases that drive the symbolic execution engine to exercise certain program paths and obtain the corresponding path constraints. A common way is to use random test cases as the initial ones [4]. Since both DSE and SPF share

the similar principle in test case generation, we implemented DSE using SPF in order to save experimental efforts. Unless otherwise specified, DSE baseline technique hereafter refers to the implementation of SPF.

For fair comparisons, each of these baseline techniques generated the same number of test cases for every object program as that of PaDMT. In addition, all baselines used random prioritization for the study of RQ3. Since our experiments involved some randomness, especially for RT and ART, each technique was repeatedly run for 30 times on every object program to guarantee the statistical reliability of our experimental results.

4.6.2. Dependent variable

The dependent variable mainly concerns about the metrics for evaluation. For RQ1, we used *mutation score* (*MS*) to examine the fault detection effectiveness of PaDMT. MS is defined as the ratio of the number of killed mutants (or revealed faults) against the total number of non-equivalent mutants (or all faults). It is formally defined as:

$$MS(P, TS) = \frac{N_k}{N_m - N_e}, \tag{4.1}$$

where P is the SUT, TS is a test suite, N_k is the number of mutants killed by TS, N_m is the total number of mutants, N_e is the number of equivalent mutants. In the context of MT, a mutant is said to be killed whenever an MR is violated in testing (i.e., an MR does not hold among the outputs of its corresponding metamorphic test group). Apparently, the higher value MS has, the more effective a testing technique is.

The *fault detection rate*, a metric similar to MS, was used for RQ2 to show the trend of fault detection as the number of test cases increases. Within a whole test suite TS for each technique on every object program, we selected the first k% test cases and measured the percentage of the number of faults detected by these test cases over the total number of faults. A higher fault detection rate intuitively implies a better effectiveness of the first k% test cases. In our study, $k = 10, 20, \ldots, 100$, where the fault detection rate when $k = 100$ is identical to MS.

For RQ3, we used the average percentage of faults detected (APFD) to evaluate the fault detection efficiency. APFD is formally defined as:

$$APFD = 1 - \frac{\sum_{i-1}^{m} TF_i}{nm} + \frac{1}{2n}, \tag{4.2}$$

Table 5: Mutation scores of source test suites for object programs

Type of Fault	Object Program	Mutation Score
Seeded Faults	BAGGAGE	73.21%
	PHONE	38.39%
	PARKING	33.55%
	CHARGE	71.16%
	TAX	97.96%
	MATH	94.71%
	Average	68.16%
Real Faults	NUMBER	37.50%

where n refers to the number of test cases in the test suite, m represents the total number of faults, and TF_i denotes the number of test cases required for detecting the ith fault. The value of APFD is between 0 and 1, and a larger APFD of a set of prioritized test cases indicates that the prioritized test cases can detect more faults with fewer test cases, and thereby the corresponding technique has a better fault detection efficiency.

For RQ4, we used the average time spent on generating a fixed number of test cases to evaluate the overhead of source test case generation. In addition, the overhead of prioritization is calculated as:

$$PO = \frac{T_{PaDMT} - T_{DSE}}{T_{PaDMT}}, \tag{4.3}$$

where T_{PaDMT} and T_{DSE} denote the time cost of PaDMT and DSE, respectively. Apparently, the smaller PO, the lower the overhead.

5. Experimental results

This section reports and analyzes the results of our experiments.

5.1. Fault Detection Effectiveness of Our Approach (RQ1)

To answer RQ1, we leveraged MS to quantitatively measure the fault detection effectiveness of test cases generated by our approach (PaDMT), as summarized in Table 5. Across the six object programs with seeded faults (that is, mutants), the average MS ranged from 33.55% to 97.96%, with a mean value of 68.16%. In a word, on average, our approach could detect nearly 70% of the faults seeded by mutation analysis.

With regard to the real faults in object program NUMBER, over one third of the real faults (3 out of 8) were detected. This observation indicates that

MT might be less effective for real faults compared with that for seeded faults. One plausible reason, as discussed in a previous study [46], is that mutation is just a simulation of real faults, and test cases that kill mutants are not guaranteed to reveal real faults. In addition, the automatically generated mutants often include a large number of "easy-to-kill" faults, whereas the real-life faults collected from the open-source projects are normally non-trivial ones as some easy-to-detect faults might already be removed before the release of a version.

More importantly, MT made use of MRs, instead of a test oracle, to verify test results. It is not surprising at all if some faults could not by reflected by MR violations. Previous studies [18] have justified that a small number of diverse MRs may be sufficient by themselves to detect most faults that are revealed by an oracle. Our evaluation results indicate that much work is yet to be done for the identification of adequate and diverse MRs to cover most functionalities/execution behaviors of SUT and thus a wide variety of faults.

5.2. Fault Detection Effectiveness: Our Approach Vs. Baselines (RQ2)

To answer RQ2, we compared the average fault detection rate of the top k% test cases ($k = 10, 20, \ldots, 100$) generated by PaDMT against those of RT, ART and DSE. Figure 4 shows the trend of fault detection rate on each object program.

Among all four techniques, PaDMT generally performed the best, followed by DSE, ART, and finally RT. The curve of PaDMT is always the first one that approaches and then reaches the upper boundary of fault detection rate. This observation indicates that our approach can enable MT to reveal as many faults as early as possible.

RT was always the worst performer in all techniques. Since all three other techniques aim at generating test cases with high fault-detection effectiveness, such an observation is consistent with our expectation. ART constantly outperformed RT, reinforcing the observation made in previous studies [21].

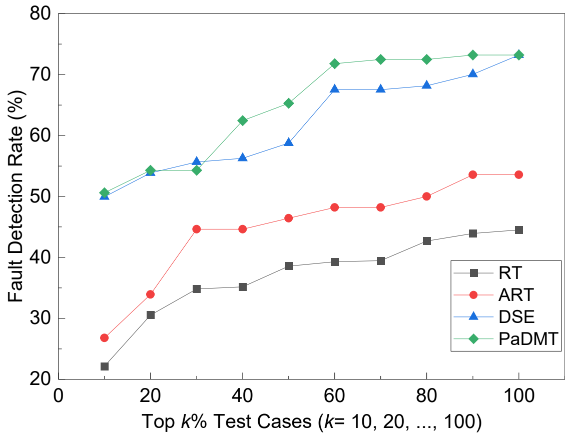
It is particularly interesting to compare the performance trend between PaDMT and DSE. In the vast majority of cases, PaDMT performed better than DSE. However, in some cases of k being small, DSE could outperform PaDMT (e.g., when $k = 30$ for BAGGAGE). As k increased, the fault detection rate of DSE was gradually approaching that of PaDMT (e.g., when $k = 100$ across all object programs). The reason is that both PaDMT and DSE exercised the maximum number of paths they could find when all the generated source test cases were exhausted. Accordingly, their fault detection effectiveness would be very close to each other with regard to the entire set of generated source test cases.

In addition, it is interesting to compare the performance trend between symbolic-execution based approaches (DSE and PaDMT) and the random approaches (RT and ART). Overall, the symbolic-execution based approaches outperformed the random approaches. When k approached 100, the fault detection rates of symbolic-execution based approaches were always better than those of the random approaches. The reason is that test cases generated by RT and ART are still random ones, for which it is very difficult to cover some "hard-to-reach" statements/paths, which are one major target of DSE and PaDMT. However, in some cases of k being small, ART could outperform DSE (e.g., when $k = 10$ for NUMBER and $k = 90$ for PHONE). In these cases, the randomness and even spreading of test cases might help random approaches quickly cover some "rare" scenarios, whereas the systematic mechanism of exploring path space limited the flexibility of symbolic execution based approaches in reaching some particular paths quickly.

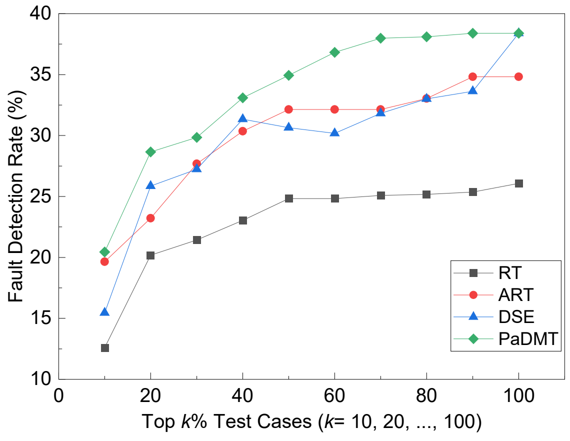
For each object program, we used SPSS to perform t-test on the fault detection rates of top k% tst cases generated by each pair of techniques (our approach vs. each benchmark technique) to test whether there is a significant difference in their averages. The results are shown in Table 6. It can be observed that the average fault detection rate of PaDMT was significantly higher than those of RT and ART ($p < 0.05$) across all values of k and object programs (except for the scenario of PaDMT versus ART for the fault detection rate of top 10% test cases of PHONE). In the vast majority of cases where the value of k was below 100, the average fault detection rate values of PaDMT were significantly higher than those of DSE across all object programs. However, when k reached 100, there was no significant difference between PaDMT and DSE.

5.3. Fault Detection Efficiency (RQ3)

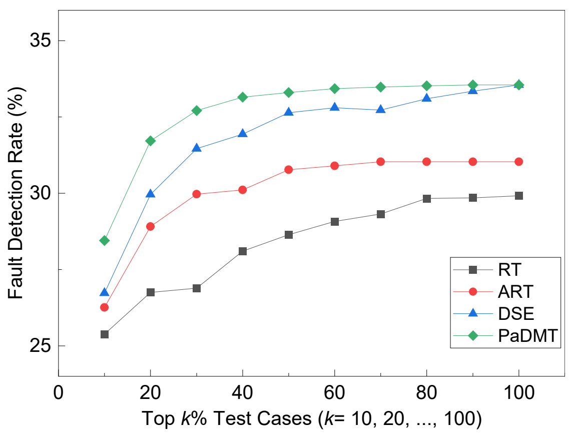
For answering RQ3, we calculated the APFD of PaDMT, in comparison with that of random prioritization of test cases generated by RT, ART, and



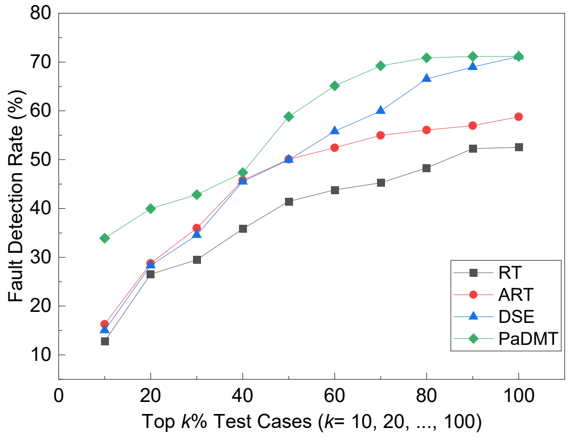
(a) BAGGAGE



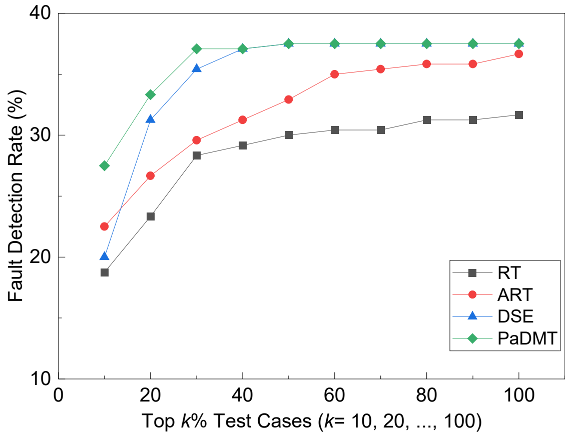
(b) PHONE



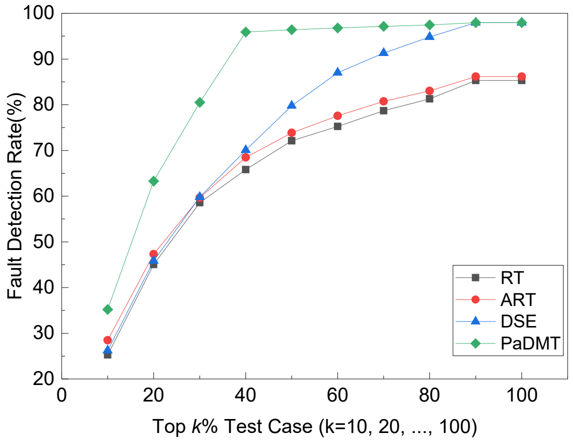
(c) PARKING



(d) CHARGE



(e) NUMBER

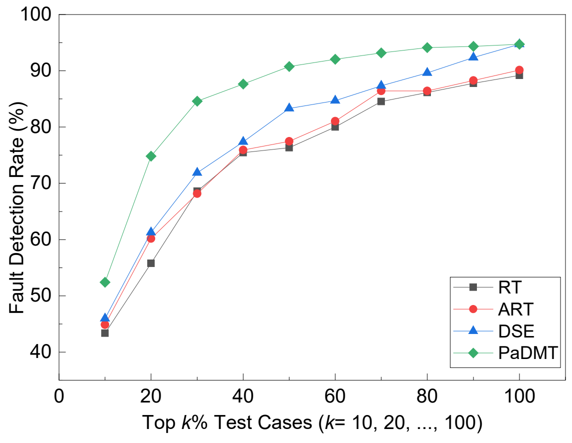


(f) TAX

Figure 4: Average fault detection rate of top $k\%$ test cases in the sequence of prioritized test cases ($k = 10, 20, \ldots, 100$)

Table 6: The P values of *t-test* on the fault detection rates of top k% test cases of pairwise techniques

Object Program	Pairwise Techniques	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%
BAGGAGE	RT vs. PaDMT	4.096×10^{-39}	8.809×10^{-14}	5.835×10^{-14}	1.010×10^{-13}	1.240×10^{-15}	1.192×10^{-20}	3.124×10^{-29}	2.384×10^{-19}	1.091×10^{-17}	6.908×10^{-15}
	ART vs. PaDMT	2.461×10^{-13}	1.469×10^{-12}	1.401×10^{-7}	1.135×10^{-8}	1.838×10^{-10}	2.948×10^{-17}	1.297×10^{-15}	8.423×10^{-16}	3.726×10^{-15}	3.385×10^{-15}
	DSE vs. PaDMT	3.089×10^{-1}	6.360×10^{-1}	7.469×10^{-1}	3.130×10^{-2}	8.415×10^{-3}	2.614×10^{-2}	1.420×10^{-2}	1.420×10^{-2}	4.340×10^{-2}	1.000
PHONE	RT vs. PaDMT	3.222×10^{-6}	3.381×10^{-8}	1.373×10^{-8}	2.199×10^{-12}	9.126×10^{-14}	1.575×10^{-21}	6.372×10^{-33}	9.836×10^{-36}	4.737×10^{-22}	5.298×10^{-25}
	ART vs. PaDMT	1.808×10^{-1}	4.247×10^{-5}	4.056×10^{-6}	2.284×10^{-2}	1.935×10^{-2}	2.360×10^{-5}	3.626×10^{-8}	7.644×10^{-8}	3.313×10^{-9}	3.313×10^{-9}
	DSE vs. PaDMT	1.205×10^{-2}	6.411×10^{-2}	4.803×10^{-2}	1.479×10^{-1}	7.130×10^{-5}	1.162×10^{-11}	1.566×10^{-9}	1.041×10^{-6}	3.109×10^{-6}	1.000
PARKING	RT vs. PaDMT	2.710×10^{-6}	5.396×10^{-11}	5.002×10^{-14}	5.197×10^{-12}	6.244×10^{-11}	6.354×10^{-11}	2.395×10^{-10}	6.529×10^{-11}	2.248×10^{-10}	3.029×10^{-9}
	ART vs. PaDMT	2.561×10^{-4}	8.381×10^{-5}	1.064×10^{-3}	9.113×10^{-5}	1.151×10^{-5}	5.078×10^{-6}	3.722×10^{-6}	2.815×10^{-6}	2.314×10^{-6}	2.314×10^{-6}
	DSE vs. PaDMT	4.224×10^{-9}	1.091×10^{-8}	7.642×10^{-8}	2.647×10^{-12}	3.333×10^{-6}	4.501×10^{-7}	6.569×10^{-6}	2.041×10^{-9}	5.656×10^{-15}	1.000
CHARGE	RT vs. PaDMT	2.419×10^{-16}	2.677×10^{-13}	5.429×10^{-14}	1.445×10^{-7}	4.307×10^{-15}	9.396×10^{-21}	1.045×10^{-22}	2.150×10^{-20}	1.087×10^{-18}	1.489×10^{-19}
	ART vs. PaDMT	6.413×10^{-12}	8.737 7×10^{-12}	1.564×10^{-5}	4.578×10^{-2}	1.824×10^{-10}	1.088×10^{-20}	9.615×10^{-24}	9.346×10^{-21}	2.773×10^{-18}	3.187×10^{-17}
	DSE vs. PaDMT	9.589×10^{-15}	6.758×10^{-8}	6.130×10^{-5}	6.006×10^{-1}	7.068×10^{-5}	3.054×10^{-6}	5.661×10^{-11}	1.520×10^{-7}	5.948×10^{-7}	1.000
NUMBER	RT vs. PaDMT	1.044×10^{-6}	7.669×10^{-7}	3.842×10^{-8}	1.503×10^{-5}	2.509×10^{-6}	1.033×10^{-6}	1.033×10^{-6}	8.699×10^{-6}	1.370×10^{-6}	2.286×10^{-5}
	ART vs. PaDMT	4.077×10^{-3}	1.508×10^{-5}	1.769×10^{-7}	1.152×10^{-5}	2.286×10^{-5}	5.913×10^{-3}	1.165×10^{-2}	2.261×10^{-2}	8.307×10^{-2}	8.307×10^{-2}
	DSE vs. PaDMT	2.886×10^{-5}	5.101×10^{-2}	5.912×10^{-3}	1.000	1.000	1.000	1.000	1.000	1.000	1.000
TAX	RT vs. PaDMT	9.333×10^{-11}	4.798×10^{-17}	1.400×10^{-24}	7.775×10^{-22}	3.749×10^{-22}	2.684×10^{-23}	2.212×10^{-20}	1.679×10^{-20}	1.478×10^{-20}	1.478×10^{-20}
	ART vs. PaDMT	2.494×10^{-7}	5.586×10^{-13}	6.744×10^{-18}	1.925×10^{-20}	2.369×10^{-21}	1.005×10^{-21}	4.064×10^{-20}	1.554×10^{-21}	1.173×10^{-24}	1.173×10^{-24}
	DSE vs. PaDMT	1.910×10^{-8}	4.916×10^{-17}	3.236×10^{-22}	2.115×10^{-22}	2.939×10^{-16}	3.768×10^{-18}	4.732×10^{-14}	6.953×10^{-8}	4.407×10^{-2}	1.000
MATH	RT vs. PaDMT	1.665×10^{-2}	1.841×10^{-5}	3.586×10^{-9}	1.186×10^{-10}	1.104×10^{-10}	1.293×10^{-9}	1.859×10^{-6}	1.160×10^{-5}	1.723×10^{-5}	2.304×10^{-4}
	ART vs. PaDMT	3.938×10^{-2}	1.418×10^{-6}	1.502×10^{-13}	6.516×10^{-8}	4.669×10^{-9}	1.268×10^{-7}	2.802×10^{-5}	2.117×10^{-6}	5.754×10^{-5}	1.163×10^{-4}
	DSE vs. PaDMT	6.336×10^{-3}	3.087×10^{-7}	1.954×10^{-8}	3.974×10^{-7}	3.627×10^{-7}	7.756×10^{-8}	2.258×10^{-8}	1.396×10^{-7}	9.611×10^{-4}	1.000



(g) MATH

Figure 4: Average fault detection rate of top k% test cases in the sequence of prioritized test cases ($k = 10, 20, \dots, 100$)

Table 7: Average APFD of prioritization techniques

Object Program	Average APFD			
	Random prioritization			PaDMT
	RT	ART	DSE	
BAGGAGE	34.89%	42.32%	58.43%	61.35%
PHONE	21.55%	28.26%	27.84%	31.75%
PARKING	26.88%	28.55%	30.15%	31.01%
CHARGE	36.20%	42.67%	46.03%	53.48%
TAX	61.73%	63.41%	69.76%	82.21%
MATH	68.09%	69.14%	75.85%	83.64%
NUMBER	26.88%	30.33%	33.00%	34.12%

DSE. The average APFD results are summarized in Table 7.

It can be observed that PaDMT achieved the highest APFD across all seven object programs, followed by DSE, ART, and RT. In six out of seven cases, DSE is better than ART. RT performed the worst for all the object programs. This observation indicates that our approach is able to reveal faults faster than the baseline techniques.

We further conducted *t-test* to verify the statistical significance of the performance difference between our technique and the other three baselines on the mean of APFD. For each baseline, the null hypothesis (H_0) was that the performance of baseline had no significant difference with that of our technique, while the alternative hypothesis (H_1) was that the performance difference was significant. The results of t-test are shown in Table 8. It can be observed that for the given confidence level $\alpha = 0.05$, the null hypothesis was rejected across all pairs of techniques and all object programs. Therefore, the average APFD difference between our technique and each of the baselines was always statistically significant.

In addition to t-test, we also calculated the effect size using *Cohen's d* to measure the magnitude of the performance difference between our technique and each of the baselines on the mean of APFD, as shown in Table 8. It can be observed that the difference between PaDMT and RT was large across all object programs (effect size > 0.8). Large difference was also observed between ART and PaDMT. Finally, the difference between DSE and PaDMT was large for six out of seven object programs, and medium for BAGGAGE (effect size > 0.5). In summary, the performance difference between our technique and the baselines was large in most cases.

5.4. Overhead of Source Test Case Generation (RQ4)

To answer RQ4, we compared the average time spent on generating a fixed number of test cases using PaDMT, RT, ART, and DSE (i.e., 34, 32, 144, 25, 52, 18, and 68 source test cases for BAGGAGE, PHONE, PARKING, CHARGE, NUMBER, TAX, and MATH, respectively). The results are shown in Table 9.

Across all object programs, the overhead of prioritization varies from 0.023% to 0.818%, which indicates that the time cost of source test case prioritization is negligible compared with that of source test case generation. Therefore, the overhead incurred by PaDMT was only marginally higher than that of DSE. Another observation is that the source test case generation time of PaDMT/DSE was much longer than that of RT/ART. It is intuitively reasonable since the symbolic execution and constraint solving can introduce heavy overhead. Nevertheless, such overhead can be negligible if we consider the long test execution time incurred by the large number of random test cases in RT/ART and the much higher fault-detection effectiveness of PaDMT.

5.5. Threats to Validity

The threats of validity of our study are discussed as follows.

Correctness of the implementation of our approach: We integrated several pieces of open-source software to support the main steps of our approach, including (1) the symbolic execution, a

Table 8: Statistical significance and effect size for the average APFD difference

Object Program	RT versus PaDMT			ART versus PaDMT			DSE versus PaDMT		
	t-test		Effect size	t-test		Effect size	t-test		Effect size
	t	p		t	p		t	p	
BAGGAGE	23.065	1.377×10^{-20}	4.211	20.856	4.891×10^{-19}	3.808	3.029	5.116×10^{-3}	0.553
PHONE	22.032	1.174×10^{-19}	2.667	4.022	8.320×10^{-6}	0.986	6.157	1.037×10^{-6}	1.124
PARKING	33.637	8.378×10^{-26}	6.141	8.609	1.758×10^{-9}	1.572	14.332	1.082×10^{-14}	2.617
CHARGE	30.275	1.700×10^{-23}	5.527	15.861	7.892×10^{-16}	2.896	6.522	3.846×10^{-7}	1.191
NUMBER	18.476	1.391×10^{-17}	3.373	6.352	6.092×10^{-7}	1.160	5.915	2.013×10^{-6}	1.080
TAX	32.470	2.553×10^{-18}	5.928	28.215	4.431×10^{-17}	5.151	18.957	1.635×10^{-14}	3.461
MATH	19.676	2.363×10^{-24}	3.592	17.692	1.229×10^{-22}	3.230	14.101	6.977×10^{-18}	2.574

Table 9: Overhead of source test case generation

Object Program	Overhead (ms)				PO(%)
	RT	ART	DSE	PaDMT	
BAGGAGE	0.161	2.115	590.266	591.756	0.252
PHONE	0.104	1.128	705.720	707.026	0.185
PARKING	0.137	1.223	2 040.505	2 043.094	0.127
CHARGE	1.421	3.165	661.279	662.565	0.194
NUMBER	0.296	0.825	691.926	697.632	0.818
TAX	0.119	0.411	1 182.843	1 185.163	0.196
MATH	0.049	0.143	11 360.578	11 363.135	0.023

main component of our approach, supported by SPF; and (2) the constraint solving supported by Choco-solver. The open-source software has been extensively used and continuously updated. In addition, other parts of the implementation have been thoroughly checked by different individuals and we are confident that their functions are in line with our requirements.

Representativeness of object programs and their faulty versions: The validity of our experimental results would be further improved if more complex object programs were included. The selection of object program was mainly due to their availability and the amount of MRs for experiments. We have collected object programs from different application domains to reduce the effect of this threat to the experimental results. With regard to the mutants of object programs, we have leveraged all applicable mutation operators to inject faults into the object programs. The number of real faults used in our experiments is relatively small, but the total amount of real faults in the open-source project is also small. Although our experiment does not involve large-size programs (e.g. those with millions of LOC), we note that it is feasible to generalize our approach to larger real-world subjects with millions of LOC with the help of following treatments: We first divide the larger real-world programs into multiple small-scale modules that can be independently tested; for each module, we analyze the corresponding MRs according to its functionality, and perform the proposed approach to derive the executable paths and their corresponding constraints; finally, source test cases are generated for each module, and the prioritization method is applied to the source test cases of each module.

Selection of baseline techniques: The comparison of source test case generation involved three baseline techniques that have used in MT field. With regard to prioritization of source test cases, we have not yet found any existing prioritization technique specifically designed for source test cases in MT. As a result, we compared our approach with random prioritization. It is still a promising research direction to extend this study by applying existing prioritization techniques in the general context of software testing into MT.

Representativeness of evaluation metrics: The evaluation metrics involved in our experiments have been extensively used in previous studies. Mutation score is a well-known metric to evaluate the fault detection effectiveness of testing techniques. APFD has been commonly used to evaluate the effectiveness of test case prioritization techniques. Thus, the threat of evaluation metrics to our experiments was minimized.

6. Related Work

Compared with the extensive investigations on MRs, a few studies have been conducted on the source test case generation for MT. Chen et al. [27] compared the performance of random source test cases and special values in MT, and observed that the test cases generated by MT are complementary to special values. Such a result was confirmed by a later study [47]. Wu [23] proposed iterative metamorphic testing (IMT) to save the efforts for source test case generation. Given a small set of initial source test cases, the existing MRs are iteratively used to generate follow-up test cases. The follow-up test cases of previous round are used as source test cases of the next round of iteration. Sun et al. [24] proposed a fixed-sized IMT technique, called FxIMT, for testing Web services with limited resources. Evaluation results showed that FxIMT exhibited a comparable fault detection effectiveness of MT, while using significantly fewer testing efforts for source test case generation and execution. Dong et al. [25] integrated MT with evolutionary testing, aiming at addressing the latter's oracle problem. In their improved evolutionary testing technique, source test cases were generated by genetic algorithms with a so-called "distance-oriented" approach — The fitness function was designed such that the generated test cases could quickly achieve all objectives such as the execution of conditions or branches. Barus et al. [21] suggested the use of adaptive random testing, an enhancement of random testing, in the source test case generation. Adaptive random testing [3] attempts to increase the diversity among test cases by evenly spreading them across the whole input domain. It was shown that such a diversity did help improve the fault-detection effectiveness of MT. The new techniques proposed in this paper also aim at improving the diversity among source test cases, from a different perspective, that is, the path distance.

Alatawi et al. [22] proposed a source test case generation strategy based on the dynamic symbolic execution (DSE), which is basically the mixture of symbolic and concrete executions. In this study, instead of DSE, we make use of the contemporary symbolic execution technique and tool (that is, SPF) to support the generation of source test cases in MT. In addition, a path-directed prioritization technique is developed to schedule the execution order of the generated test cases. Also provided are a complete framework and an automated tool that systemically integrate all proposed techniques into MT.

The integration of symbolic execution and MT was first proposed by Chen et al. [48]. In their "semi-proving" approach, MT was applied to alleviate the oracle problem in program proving. Symbolic inputs, instead of concrete values, were used as the test cases for MT. In this way, a satisfaction of an MR on symbolic test cases would guarantee the correctness of SUT on certain properties, that is, the program could be semi-proven. By contrast, our study still uses concrete test cases, although their generation is guided by symbolic execution. This is due to the different context we are targeting at — Our goal is to improve the performance of MT.

7. Conclusion

Metamorphic testing (MT) is a simple yet effective technique that not only alleviates the oracle problem effectively, but also constructs test cases that are complementary to those created by traditional testing methods. In addition to the metamorphic relations (MRs), the generation of source test cases attracted increasing research interests. Some techniques have been proposed to generate "good" source test cases that help improve the overall performance of MT. In this paper, we developed a new path-directed method for the source test case generation. It utilizes the techniques of symbolic execution and constraint solver to obtain program path constraints, which, in turn, provide the basis for generating source test cases that achieve a good coverage of execution paths and thus deliver a higher fault detection effectiveness. An additional prioritization technique was proposed to further improve the diversity among test cases and hence boost the testing efficiency. A tool was developed to automate the new techniques and integrate them with the existing MT4WS tool. The experimental studies based on seven representative programs demonstrated the high performance of the proposed techniques.

The study reveals quite a few research directions for future work. We plan to study the performance of our approach and tool through the application into industrial large-size programs in the future. For example, more large-scale empirical studies are necessary to further evaluate the performance of the proposed path-directed techniques. It is also interesting to investigate how to integrate other types of symbolic execution, such as dynamic symbolic execution and symbolic backward execution, into our tool. The present study only made use of path constraints in the process of source test case generation. It is worthwhile to study whether and how the concept can be used in the construction of follow-up test cases. A promising direction is to consider more sophisticated distance measures and coverage criteria discussed in [49] for prioritizing the source test cases, with an aim to further improve the fault detection efficiency of MT. In addition, compared with random prioritization, it is worthwhile to study the improvement of fault detection efficiency achieved by different prioritization strategies. Finally, it is of importance to study the difference between the cost-effectiveness of manually identifying MRs with that of manually defining test oracles, which helps to demonstrate the practical benefits of MT.

Acknowledgements

This research is supported by the National Natural Science Foundation of China under Grant Nos. 61872039 and 61370061, the Fundamental Research Funds for the Central Universities (Grant No. FRF-GF-19-B19), the Beijing Natural Science Foundation (Grant No. 4162040), the Aeronautical Science Foundation of China (Grant No. 2016ZD74004), and the Australia Research Council Discovery Project (Grant No. DP210102447).

References

[1] J. Buxton, B. Randell, Report on a conference sponsored by the nato science committee, Tech. rep., NATO Science Committee (1970).

[2] T. Y. Chen, T. H. Tse, Z. Q. Zhou, Fault-based testing without the need of oracles, Information and Software Technology 45 (1) (2003) 1–9.

[3] A. C. Barus, T. Y. Chen, F.-C. Kuo, H. Liu, R. Merkel, G. Rothermel, A cost-effective random testing method for programs with non-numeric inputs, IEEE Transactions on Computers 65 (12) (2016) 3509–3523.

[4] P. Godefroid, N. Klarlund, K. Sen, DART: Directed automated random testing, in: Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI'05), ACM, 2005, pp. 213–223.

[5] C.-A. Sun, H. Dai, H. Liu, T. Y. Chen, K.-Y. Cai, Adaptive partition testing, IEEE Transactions on Computers 68 (2) (2019) 157–169.

[6] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. Tse, Z. Q. Zhou, Metamorphic testing: A review of challenges and opportunities, ACM Computing Surveys 51 (1) (2018) 4:1–4:27.

[7] S. Segura, G. Fraser, A. B. Sanchez, A. Ruiz-Cortés, A survey on metamorphic testing, IEEE Transactions on Software Engineering 42 (9) (2016) 805–824.

[8] Y. Tian, K. Pei, S. Jana, B. Ray, DeepTest: Automated testing of deep-neural-network-driven autonomous cars, in: Proceedings of the 40th International Conference on Software Engineering (ICSE'18), 2018, pp. 303–314.

[9] Z. Q. Zhou, L. Sun, Metamorphic testing of driverless cars, Communications of the ACM 62 (3) (2019) 61–67.

[10] X. Xie, J. W. K. Ho, C. Murphy, G. E. Kaiser, B. Xu, T. Y. Chen, Testing and validating machine learning classifiers by metamorphic testing, Journal of Systems Software 84 (4) (2011) 544–558.

[11] V. Le, M. Afshari, Z. Su, Compiler validation via equivalence modulo inputs, in: ACM SIGPLAN Notices, Vol. 49, 2014, pp. 216–226.

[12] T. Y. Chen, P.-L. Poon, X. Xie, METRIC: Metamorphic relation identification based on the category-choice framework, Journal of Systems and Software 116 (2016) 177–190.

[13] C.-A. Sun, A. Fu, P.-L. Poon, X. Xie, T. Y. Chen, METRIC+: A metamorphic relation identification technique based on input plus output domains, IEEE Transactions on Software Engineering (2019).
URL https://doi.org/10.1109/TSE.2019.2934848

[14] S. Segura, A. Durán, J. Troya, A. Ruiz-Cortés, Metamorphic relation patterns for query-based systems, in: Proceedings of the 4th International Workshop on Metamorphic Testing (MET'19), Co-located with the 41th International Conference on Software Engineering (ICSE'19), 2019, pp. 24–31.

[15] B. Zhang, H. Zhang, J. Chen, D. Hao, P. Moscato, AutoMR: Automatic discovery and cleansing of numerical metamorphic relations, in: Proceedings of the 2019 IEEE International Conference on Software Maintenance and Evolution (ICSME'19), 2019, pp. 235–245.

[16] S. Segura, J. A. Parejo, J. Troya, A. Ruiz-Cortés, Metamorphic testing of RESTful web APIs, IEEE Transactions on Software Engineering 44 (11) (2018) 1083–1099.

[17] Y. Cao, Z. Q. Zhou, T. Y. Chen, On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions, in: Proceedings of the 13th International Conference on Quality Software (QSIC'13), 2013, pp. 153–162.

[18] H. Liu, F.-C. Kuo, D. Towey, T. Y. Chen, How effectively does metamorphic testing alleviate the oracle problem?, IEEE Transactions on Software Engineering 40 (1) (2014) 4–22.

[19] J. Mayer, R. Guderlei, An empirical study on the selection of good metamorphic relations, in: Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC'06), 2006, pp. 475–484.

[20] T. J. Ostrand, M. J. Balcer, The category-partition method for specifying and generating fuctional tests, Communications of the ACM 31 (6) (1988) 676–686.

[21] A. C. Barus, T. Y. Chen, F.-C. Kuo, H. Liu, H. W. Schmidt, The impact of source test case selection on the effectiveness of metamorphic testing, in: Proceedings of the 1st International Workshop on Metamorphic Testing (MET'16), Co-located with the 38th International Conference on Software Engineering (ICSE'16), 2016, pp. 5–11.

[22] E. Alatawi, T. Miller, H. Søndergaard, Generating

source inputs for metamorphic testing using dynamic symbolic execution, in: Proceedings of the 1st International Workshop on Metamorphic Testing (MET'16), Co-located with the 38th International Conference on Software Engineering (ICSE'16), 2016, pp. 19–25.

[23] P. Wu, Iterative metamorphic testing, in: Proceedings of the 29th Annual International Computer Software and Applications Conference (COMPSAC'05), 2005, pp. 19–24.

[24] C.-A. Sun, A. Fu, Z. Wang, Q. Wen, P. Wu, T. Y. Chen, Iterative metamorphic testing for web services: Technique and case studies, International Journal of Web and Grid Services 16 (4) (2020) 364–392.

[25] G. Dong, S. Wu, G. Wang, T. Guo, Y. Huang, Security assurance with metamorphic testing and genetic algorithm, in: Proceedings of the IEEE/WIC/ACM International Conference on Web Intelligence and Intelligent Agent Technology (WI-IAT'10), Vol. 3, 2010, pp. 397–401.

[26] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, I. Finocchi, A survey of symbolic execution techniques, ACM Computing Surveys 51 (3) (2018) 50:1–50:39.

[27] T. Y. Chen, F.-C. Kuo, Y. Liu, A. Tang, Metamorphic testing and testing with special values, in: Proceedings of the 5th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD'04), 2004, pp. 128–134.

[28] T. Y. Chen, J. W. Ho, H. Liu, X. Xie, An innovative approach for testing bioinformatics programs using metamorphic testing, BMC bioinformatics 10 (1) (2009) 24–32.

[29] K. Sen, G. Agha, CUTE and jCUTE: Concolic Unit Testing and Explicit Path Model-Checking Tools, in: Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06), 2006, pp. 419–423.

[30] C. S. Păsăreanu, N. Rungta, Symbolic PathFinder: Symbolic execution of Java bytecode, in: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE'10), ACM, 2010, pp. 179–180.

[31] C. S. Păsăreanu, W. Visser, D. Bushnell, J. Geldenhuys, P. Mehlitz, N. Rungta, Symbolic PathFinder: integrating symbolic execution with model checking for Java bytecode analysis, Automated Software Engineering 20 (2013) 391–425.

[32] W. Visser, C. S. Păsăreanu, S. Khurshid, Test input generation with Java PathFinder, in: Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'04), 2004, pp. 97–107.

[33] C. S. Păsăreanu, N. Rungta, W. Visser, Symbolic execution with mixed concrete symbolic solving, in: Proceedings of the 2011 International Symposium on Software Testing and Analysis (ISSTA'11), 2011, pp. 34–43.

[34] S. Yoo, M. Harman, Regression testing minimization, selection and prioritization: A survey, Software Testing, Verification and Reliability 22 (2) (2012) 67–120.

[35] G. Rothermel, R. H. Untch, C. Chu, M. J. Harrold, Prioritizing test cases for regression testing, IEEE Transactions on Software Engineering 27 (10) (2001) 929–948.

[36] Symbolic pathfinder, last accessed on October 11, 2020.
URL `https://github.com/SymbolicPathFinder/jpf-symbc`

[37] K. Havelund, T. Pressburger, Model checking Java programs using Java PathFinder, International Journal on Software Tools for Technology Transfer 2 (2000) 366–381.

[38] N. Jussien, G. Rochart, X. Lorca, Choco: An open source Java constraint programming library, in: CPAIOR'08 Workshop on Open-Source Software for Integer and Contraint Programming (OSSICP'08), 2008, pp. 1–10.
URL `https://hal.archives-ouvertes.fr/hal-00483090`

[39] Choco-solver, last accessed on October 12, 2020.
URL `https://choco-solver.org/`

[40] D. Kroening, O. Strichman, Decision Procedures, Springer, 2016.

[41] C.-A. Sun, G. Wang, Q. Wen, D. Towey, T. Y. Chen, MT4WS: An automated metamorphic testing system for web services, International Journal of High Performance Computing and Networking 9 (1-2) (2016) 104–115.

[42] C.-A. Sun, Y. Liu, Z. Wang, W. Chan, μMT: A data mutation directed metamorphic relation acquisition methodology, in: 2016 IEEE/ACM the 1st International Workshop on Metamorphic Testing (MET'16), Co-located with the 38th International Conference on Software Engineering (ICSE'16), 2016, pp. 12–18.

[43] R. Just, D. Jalali, M. D. Ernst, Defects4J: A database of existing faults to enable controlled testing studies for Java programs, in: Proceedings of the 2014 International Symposium on Software Testing and Analysis (ISSTA'14), ACM, 2014, pp. 437–440.

[44] M. Papadakis, M. Kintis, J. Zhang, Y. Jia, Y. Le Traon, M. Harman, Mutation testing advances: An analysis and survey, Advances in Computers (2017).
URL `http://orbilu.uni.lu/bitstream/10993/31612/1/survey.pdf`

[45] Y.-S. Ma, J. Offutt, Y.-R. Kwon, muJava: A mutation system for Java, in: Proceedings of the 28th International Conference on Software Engineering (ICSE 2006), ACM, 2006, pp. 827–830.

[46] D. Shin, S. Yoo, D.-H. Bae, A theoretical and empirical study of diversity-aware mutation adequacy criterion, IEEE Transactions on Software Engineering 44 (10) (2018) 914–931.

[47] P. Wu, X.-C. Shi, J.-J. Tang, H.-M. Lin, T. Y. Chen, Metamorphic testing and special case testing: A case study, Journal of Software 16 (7) (2005) 1210–1220.

[48] T. Y. Chen, T. H. Tse, Z. Zhou, Semi-proving: An integrated method for program proving, testing, and debugging, IEEE Transactions on Software Engineering 37 (1) (2011) 109–125.

[49] P. Saha, U. Kanewala, Fault detection effectiveness of source test case generation strategies for metamorphic testing, in: Proceedings of the 3rd International Workshop on Metamorphic Testing (MET'18), Co-located with the 40th International Conference on Software Engineering (ICSE'18), ACM, 2018, pp. 2–9.

Conflict of Interest Statement

We declare that we do not have any commercial or associative interest that represents a conflict of interest in connection with the work submitted.

Chang-ai Sun, Baoli Liu, An Fu, Yiqiang Liu, and Huai Liu