Using Metamorphic Testing to Improve Dynamic Symbolic Execution

Eman Alatawi  
Computing and Information Systems  
The University of Melbourne  
Victoria 3010, Australia  
Email: e.alatawi@student.unimelb.edu.au

Tim Miller  
Computing and Information Systems  
The University of Melbourne  
Victoria 3010, Australia  
Email: tmiller@unimelb.edu.au

Harald Søndergaard  
Computing and Information Systems  
The University of Melbourne  
Victoria 3010, Australia  
Email: harald@unimelb.edu.au

Abstract—Dynamic symbolic execution (DSE) is an approach for automatically generating test inputs from source code using constraint information. It is used in fuzzing; the execution of tests while monitoring for generic properties such as buffer overflows and other security violations. Limitations of DSE for fuzzing are two-fold: (1) only generic properties are checked; many deviations from specified behaviour are not found; and (2) many programs are not entirely amenable to DSE because they give rise to hard constraints, so that some parts of a program remain uncovered. In this paper, we discuss how to mitigate these problems using metamorphic testing (MT). Metamorphic testing uses domain-specific properties about program behaviour, relating pairs of inputs to pairs of outputs. From a given test suite, follow-up tests are generated, and their outputs are compared to outputs from the original tests, using metamorphic relations. Our hypothesis is that using metamorphic testing increases the ability of a DSE test suite to find faults, and that the follow-up tests execute some previously-uncovered segments. We have experimented with seven small but non-trivial libraries, comparing DSE test suites with DSE+MT test suites, demonstrating that DSE+MT test suites improve coverage marginally, but find more faults.

I. INTRODUCTION

A critical challenge in software testing is to achieve adequate testing beyond the limited scope of manual testing. Several techniques have been proposed to automatically generate test cases that help test software systems extensively [1]. Dynamic symbolic execution (DSE) [2] has been proposed as one way to generate tests automatically from source code. Given a program and an input, DSE uses symbolic execution [3] to record how the input affects the control flow in the program. The result is a path constraint: a logic formula that is satisfied by the input and also by any other input that will drive the program’s execution along the same execution path. It mutates this formula to provide test inputs for paths that are (at least partially) uncovered by previous tests, and repeats this process until a specified form of coverage is achieved.

Recent work [2], [4] has extended the concept of DSE to whitebox fuzzing, in which DSE is used to generate large amounts of system-level tests for testing security properties. While this use for DSE has proved valuable at finding such security faults (e.g., see Godefroid et al. [4]), one of the major limitations of fuzzing is that it detects only generic faults such as exceptions, buffer overflows, or violation of specific security properties; or it requires assertions in the program’s code [1]. The approach does not reveal faults that are domain specific. For that, a test oracle would be needed, and when available, it can increase the value of running a DSE tool. However, in practice, good test oracles are rarely available. Hence, this task is often done manually, if at all. Moreover, while DSE tools often cover a large fraction of a program’s code, limitations in constraint solving technology mean that some parts of the program may remain untested.

Studies that investigate the fault-finding effectiveness of DSE approach against other methods of test generation are few. Most studies of DSE focus on guiding the test generation process to cover as many program paths as possible to achieve high code coverage using a specific criterion, such as branch coverage. However, several studies, including a recent large-scale investigation by Inozemtseva et al. [5], suggest that code coverage, which is the main objective of DSE tools, is not a sufficient indicator of the ability of test suites to uncover faults. For this reason, the effectiveness, in practice, of DSE could be questioned. In addition, although the test oracle is a significant issue when using code-based test generators and the choice of oracle can have a crucial impact on the effectiveness of the overall testing process [6], there is no study investigating the influence of the used test oracle on the fault-finding effectiveness of test suites generated using DSE.

In this paper, we experimentally investigate the extent to which test oracles can increase the value of DSE. We particularly integrate DSE test generation with Metamorphic Testing (MT) [7], [8]. MT is a technique that aims to mitigate the problem of oracle unavailability in software testing by using existing test suites (source tests) to generate follow-up tests based on a set of required properties of the targeted functionality of the program. These properties, called metamorphic relations (MRs) are expressed as a relationship between the program inputs, and their corresponding outputs. When a program is found to violate one of these relations, it signals an error in the program or the relation itself.

We are interested in these two questions:

Q1 To what extent can metamorphic testing increase the ability to uncover defects in test suites generated by DSE in the absence of the test oracle?
Q2 Can the follow-up tests generated using metamorphic testing be used to cover parts of the program that are otherwise left uncovered using DSE?

We ran experiments using seven collections of programs from different application areas. We generated test suites from the programs using two DSE tools, jCUTE [9] and Pex [10], and we used mutation analysis to estimate the fault-finding effectiveness of the test suites under three conditions: (a) using no test oracle; (b) using a metamorphic test oracle; and (c) using a “golden” test oracle, which is the original, non-mutated version of the program, as an upper baseline. Because metamorphic testing creates additional test inputs, we also controlled for the increased test suite size by running the extended test suite using all three oracles.

Our findings can be summarized as follows:

- DSE with no oracle is remarkably good at finding a large amount of mutants, finding over 40% of the faults found using a golden oracle.
- Extending the test suites with metamorphic tests offered a negligible increase in branch coverage overall, but also significantly improving coverage in programs on which DSE is not so effective.
- Using metamorphic relations as functional test oracles increase the mutant score by approximately 14% over no oracle, providing evidence to support that hypothesis that metamorphic testing can be used effectively to increase the value of DSE.
- While DSE was able to achieve high code coverage, this did not correlate with a high mutation score, indicating that code coverage should not be used as a sole criteria to judge DSE tools.
- In the absence of an oracle, MT provides a disciplined approach to test data generation.

These results suggest that DSE provides value to testers, but that the addition of good test oracles can be useful.

II. BACKGROUND AND RELATED WORK

In this section, we outline the necessary background on dynamic symbolic execution (DSE) and test oracles.

A. Dynamic Symbolic Execution (DSE)

According to Sen et al. [9], the idea of extending classical SE with additional run-time information by mixing concrete and symbolic executions was first proposed by Larson and Austin [11]. Larson et. al’s goal was to improve the coverage by executing the program on some given concrete inputs, and by solving the symbolic constraints collected along a specific execution to find any input that could cause a violation in that specific path. This idea created a wide range of novel research including variant approaches aiming to explore not only specific paths but to explore as many as possible of feasible paths of the program and use that to automatically generate effective and comprehensive test inputs.

DSE consists of executing a program in two different ways simultaneously, starting with some given concrete input, while also symbolically executing the same program path, collecting and storing symbolic constraints about the input at branch points along the way. Once a path has been executed, the constraints collected form a path constraint for that path, which explicate (or approximate) the conditions on the input that caused that path to be executed. By selectively negating one or more of these constraints and requesting a concrete solution for the modified constraint using a constraint solver, a new input can be generated, which will direct the execution towards a different path in the program. This process repeats until some pre-determined criteria, such as branch coverage, are met.

A difficulty with DSE is the complexity of reasoning about constraints. For example, a branch guarded by a non-linear test may generate a constraint which cannot be solved by the underlying constraint solver. While DSE can mitigate this problem in some cases by using the concrete value of a variable in place of its symbolic value, this solution does not work for all cases.

Several variants of DSE have been proposed, such as Directed Automated Random Testing (DART) that combines DSE with random testing and model checking techniques [12], and concolic testing that extends DART to handle dynamic data structures such as pointers and multi-threaded programs [9], [13].

B. Test Oracles

An oracle is a mechanism used to determine whether the program under test is working as intended by comparing the actual values against the expected values or specified properties [6]. It consists of two parts: oracle information that defines the software’s expected output, and an oracle procedure that specifies how to compare the expected output against the actual output [14]. Obtaining such a mechanism is difficult and expensive in most cases (e.g., generate expected information for file directories, machine registers, or memory, etc), and it may be unavailable altogether [8]. This is the oracle problem, a well-known issue in software testing and programs without oracles are sometimes referred to as “non-testable programs” [15].

However, while effort has been put into automating many aspects of software testing, the problem of constructing test oracles automatically or even semi-automatically appears to be neglected, despite its importance [16]. Even with the recent emergent interest in finding new techniques and concepts to solve the test oracle problem and to reduce human oracle cost [17], the lack of test oracles remain an obstacle for a complete and effective test automation. Current research in test oracles aims to either assist testers in selecting effective oracle information (i.e. semi-automated test oracles), or construct test oracles automatically [18]. Based on the source of oracle, the used automation method, and the type of software output, there are different taxonomies for test oracles [14]. Harman et al. [17] classify test oracles into three categories: specified oracles, implicit oracles, and derived oracles. They define specified oracles as those based on a formal specification of a system, such as algebraic specifications, model-based...
specifications, or contracts (e.g., [19], [20], [21]). Derived 
oracles are those based on system artifacts, such as document-
tation, domain-specific properties (e.g. metamorphic relations),
or previous versions of systems. Implicit oracles are those that 
capture generic properties, such as exceptions, deadlocks, 
violations of security properties, or abnormal termination.

**Test oracles in DSE.** DSE is often applied without a test 
oracle, as a fuzzing tool [2], [4]. In such cases, DSE is used 
to generate and run a large set of inputs, while monitoring the 
execution for **generic** properties such as uncaught exceptions 
and buffer overflows, or while running a security monitor. De-
spite the lack of a test oracle, whitebox fuzzing has been used in 
industry with strong outcomes; for example, Bounimova et al. [4] report that their SAGE tool, developed internally at 
Microsoft, detected one-third of all security vulnerabilities in 
Windows 7 during its development, saving millions of dollars 
[4]. In this work, we consider whether DSE effectiveness could 
be improved by the inclusion of checking some functional 
properties of programs by introducing derived oracles in the 
form of metamorphic relations.

### C. Metamorphic Testing

Conventional testing considers test cases worthless when 
they fail to reveal a failure in the software. However, it is 
possible to utilise the information that these test cases carry, 
to alleviate the test oracle problem. Metamorphic Testing (MT) 
aims to do just that by using metamorphic relations as a core 
part for both test case generation and test result verification 
[22], [23].

MT is a systematic and arguably cost-effective way to verify 
test outputs based on a set of metamorphic relations. The idea 
is that, when it might be impossible to judge the correctness of 
software output according to a given input, that software still 
exhibits some properties that can help to predict the output 
resulting from a systematic transformation of the given input 
[24].

Basic steps of MT outlined in [22], [14] are as follows:

1. Derive a set of **metamorphic relations** (MRs) that specify 
a relationship between two or more program inputs, and 
the corresponding outputs, that should be satisfied by 
the software under test (i.e. oracle information).

2. Generate a set of test cases (called **source tests**) using 
any test generation approach (e.g., manual or random 
test input generation).

3. Generate **follow up tests** by transforming source tests 
generated in step 2, based on the MRs defined in step 1.

4. Execute the source and follow-up tests to check whether 
the change in the output agrees with the predicted 
change specified by the MR.

More precisely, given a program under test, \( p \), inputs \( i_1 \) and 
\( i_2 \), a relation between inputs for \( p, \sim_i \), and a relation between 
the outputs of those inputs, \( \sim_o \), MT can be used to verify the 
outputs of \( i_1 \) and \( i_2 \) if the following holds:

\[
i_1 \sim_i i_2 \rightarrow p(i_1) \sim_o p(i_2)
\]

A testing process executes the source test, \( i_1 \), generates a 
follow-up test, \( i_2 \) using \( \sim_i \), runs \( p \) on both inputs, and checks 
whether the outputs are related using \( \sim_o \). If they are not, this 
represents a fault in the program (or alternatively, a mistake 
in the metamorphic relations themselves!)

As an example, consider a function that sorts a sequence of 
integers in ascending order. Given source test input – an array 
of integers – we can generate a follow-up test by randomly 
permuting the elements of the array. When we run the program 
on both inputs, the outputs should to be identical. In this 
case, the input relation, \( \sim_i \), is permutation, while the output 
relation, \( \sim_o \), is the identity relation. Although MT checks 
the correctness of the relation among multiple executions 
rather than independent test outputs, it can be considered as an 
(automated) derived oracle.

The MR can be any sort of relation: equality, inequality, 
subsumption, convergence, as well as all kind of numeric 
relations. In addition, there are general MRs that can be used 
for specific software domains, and proposed guidelines for 
selecting good relations based on the specific domain of the 
software. These guidelines have been applied successfully in 
different domains [24].

Consider the different properties identified for the sort 
function, as shown in Figure 1. The addition of a random 
positive integer \( c > 0 \) to each element in the source test input 
\( x = [x_1, x_2, \ldots, x_n] \) \((n > 0)\) generates follow-up test input 
\( x' = [x_1 + c, x_2 + c, \ldots, x_n + c] \), and the output for the follow-
up test input should be identical to that for the original test 
input, except that \( c \) has been added to each element. In this 
case, the input relation is:

\[
\exists c : \left( c > 0 \land \right) \left( x = [x_1, \ldots, x_n] \land x' = [x_1 + c, \ldots, x_n + c] \land \forall i \in [1, n] : \sim \text{overflow}(x_i + c) \right)
\]

and a useful output relation is:

\[
\text{sort}(x) \neq \text{sort}(x') \land \forall i \in [1, n] : (x[i] + c = x'[i])
\]

(whether the sorting algorithm is stable or not).

Similarly, if source test input \( x = [x_1, x_2, \ldots, x_n] \) of 
length \( n \) is reversed to generate the follow-up test input 
\( x' = [x_n, \ldots, x_1] \), then the output for the follow-up test input 
should be identical to that for the source test input. In this 
case, the relevant input-output relation is:

\[
x' = \text{reverse}(x) \Rightarrow \text{sort}(x) = \text{sort}(x')
\]

**Derivation of MRs.** Constructing metamorphic relations that 
are able to reveal software faults is the main challenge 
in MT, and still an open research question. They tend to be 
derived manually by testers in an ad hoc manner as there is no 
formal methodology for identifying MRs in a systematic way 
[23]. Many studies showed that deriving MRs is not a difficult 
task, and they provide some general guidelines to help testers 
[25]. For example, there are different classes of metamorphic 
properties that can be applied to derive MRs for functions 
and applications that deal with numerical inputs and outputs,
such as multiplying numerical values of a source test input by a constant, permuting the order of the elements in a source test input, taking the inverse of each element, adding a new element, removing an element, or creating a set from some number of smaller sets of source test inputs [24].

Generally, to derive MRs, testers need to consider the properties of similar applications in the given domain, the properties of the chosen algorithms to solve the given problem in that domain, the properties of the implementation of that algorithm taking into account the used programming language [24], [26]. It is recommended also to avoid the metamorphic relations that contain similar computations as the implemented algorithm. Not surprisingly, MRs that have rich semantic properties tend to detect more faults than simple ones [25].

Automated discovery of metamorphic properties is a challenge in metamorphic testing that is still at an early stage of investigation [17]. The automated approach of discovery MRs can be built upon existing techniques such as dynamic discovering of program invariants, algebraic specifications, or by using machine learning techniques [24], [14].

Applicability and research directions. Standard examples of MT use numeric relations. However, MT can be applied to non-numerical applications domains such as bioinformatics [27], search engines [28], machine learning [29], and web services [30].

Generally, research directions in MT can be classified into two categories: (1) studies that aim to apply MT on testing software from different domains to alleviate the test oracle problem, and (2) studies that aim to improve the technique itself of MT to be applied in different contexts in different ways (e.g., fault localization). This demonstrates an evolution of research on MT that represent a visible trend in test oracles literature [14].

In this paper, we use MT to alleviate the test oracle problem in DSE, investigating to what extent this can increase the effectiveness of DSE in finding more faults. The source test inputs play a vital role in the whole MT process, thus; integrating MT with DSE as test inputs generation technique can be beneficial for MT as well. Our approach is different from the semi-proving method presented by Chen et al. [31], where symbolic execution (not DSE) is used to derive symbolic inputs that are then used to prove selected MRs in a given program. Instead, we integrate MT and DSE to improve testing by increasing the effectiveness of DSE even in the absence of complete test oracles.

D. Effectiveness of DSE

While much research has been dedicated to find solutions to address DSE challenges, there are few studies that evaluate the DSE approach, comparing it to other automatic test generation approaches. In addition, there are no standard metrics for evaluating the effectiveness.

The main goal of testing is to reveal a maximum number of faults in a program within a given budget. Typically, most studies use code coverage (which is the goal of DSE tools) to evaluate the quality of test suites, assuming the more code covered by the test suites, the more effective they are. However, few studies investigate the correlation between code coverage and the number of detected faults [1]. There is also some inconsistency in the results of these studies. While some show that high coverage indicates a higher probability to detect real faults (e.g., [32], [33]), others have found that code coverage is not a sufficient or a reliable indicator of the effectiveness of the test suites (e.g., [34], [5], [35]). Furthermore, Kim et al. [36] assert that standard coverage metrics formulated to help testers manually generate test suites need to be updated now that automated test generation is a reality.

Table 1 summarises the findings of the available studies that have investigated the effectiveness of DSE tools, and compared them to other approaches.

<table>
<thead>
<tr>
<th></th>
<th>DSE generated test cases</th>
<th>Metamorphic relation used</th>
<th>MR follow-up test cases</th>
<th>Checking the relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X = [1, 0], \ldots$</td>
<td>MR1. Permutation</td>
<td>$X' = \tau_1(X) = [0, 1]$</td>
<td>$\checkmark$</td>
<td>$\checkmark$</td>
</tr>
<tr>
<td>$X = [1, 0], \ldots$</td>
<td>MR2. Addition</td>
<td>$X' = \tau_2(X) = [5, 4]$</td>
<td>$\checkmark$</td>
<td>$\checkmark$ \land X[i] &lt; X'[i]</td>
</tr>
<tr>
<td>$X = [1, 0], \ldots$</td>
<td>MR3. Multiplication</td>
<td>$X' = \tau_3(X) = [2, 0]$</td>
<td>$\checkmark$</td>
<td>$\checkmark$ \land X[i] \leq X'[i]</td>
</tr>
<tr>
<td>$X = [1, 0], \ldots$</td>
<td>MR4. Extension</td>
<td>$X' = \tau_4(X) = [1, 0, 2]$</td>
<td>$\checkmark$</td>
<td>$\checkmark$ sort(X) \neq sort(X')</td>
</tr>
</tbody>
</table>

Fig. 1. Metamorphic relations for a sorting program
in addition to the number of detected faults, such as boundary or mutation coverage, logical coverage (Jamrozik et al. [39]), and label coverage (Bardin et al. [40]). Many recent studies draw a similar conclusion that code coverage is not the best measure, and the focus should be on fault detection ability and exploring practical criteria that provide a better indication of the effectiveness of test suites. Staats et al. [34] suggest considering code coverage should come after generating test suites to identify the parts of the code that have not been tested, rather than as an explicit goal.

### III. USING METAMORPHIC TESTS IN DSE

To improve the effectiveness of DSE generated test suites without incurring a high cost, we propose combining DSE and metamorphic testing (MT) to detect domain-specific program faults. Thus, we combine DSE’s ability to detect generic faults (and assertion violations, if used in the code) with high code coverage, and MT’s ability to detect faults related to functional properties of programs.

As far as we know, this has not been investigated before, and we assert that this can improve the value of DSE by extending its ability to reveal faults. Further to this, our hypothesis is that the follow-up test cases have the potential to cover some parts of the program that DSE cannot reach due to its inherent limitations, such as difficulties in reasoning about non-linear constraints.

Starting with a source test suite with \( n \) DSE generated test inputs, we generate an additional follow-up test suite of at most size \( n \times m \) test inputs, where \( m \) is the number MRs identified from the program domain. The source test suite aims to cover as many as possible of the program paths using a minimal number of test cases, while the follow-up test cases can be used to improve coverage.

To demonstrate how MT-enhanced DSE can be effective, consider again the example of a sorting algorithm. To perform unit testing for this function, assume that a DSE tool generated a source test suite \( S \) of size 4 and branch coverage 100%; for example, \( S \) may be \( \{1, 0, 0, 0, 1, 0\} \). The DSE test suite in this case covers all branches of the program for both the true and false cases. On its own, this test suite may detect uncaught exceptions and program cases. After executing this test input, we should verify correctness, and the only way to do that is to have a test oracle that checks that output is correct; that is, the output sequence is in ascending order, and the input and output are identical when seen as multisets.

Assume that we introduce a fault into a bubble sort implementation, so that this program fragment does not swap items when it should. A DSE tool such as jCUTE or Pex will still achieve 100% branch coverage, however, all tests from that test suite will pass because there are no generic violations. Assuming a perfect oracle, the test input \( [1, 0] \) will enable detection of the fault. Clearly, code coverage in this case does not reflect the test suite effectiveness in terms of fault detection capability.

To see how MT can improve this, consider a follow up test in which a source input array \( x \) is rearranged in reverse order to generate another test input \( x' = [1, 0] \). This permutation MR says that, if \( x' = \text{reverse}(x) \), then \( \text{sort}(x) = \text{sort}(x') \). This relation holds for all executions. Thus, if the output of \( \text{sort}(x) \) and \( \text{sort}(x') \) are different, a fault is detected by that test. This way the seeded fault described above would be detected.

Given that metamorphic relations can be programmed, and the change of outputs between source and follow-up test inputs is predictable [26], [24], the process of integrating DSE

\(^1\)Indeed this is the test suite generated by jCUTE.
with MT can be automated, and then DSE can be applied automatically to test programs for functional properties.

IV. EXPERIMENT DESIGN

This section presents the design of our experiment and illustrates the experimental procedure.

We collected several well-known small, but non-trivial programs, and generated tests from these using two DSE tools: jCUTE [9] and Pex [10]. We then extended the resulting test suites using metamorphic relations specific to the programs, and measured the coverage of the original and extended test suites. We measured the mutation score of these test suites, and measured the coverage on the program. ‘#Types’ refers to the number of types used in the program, which is of interest because most DSE tools can consider only a handful of basic types in their symbolic reasoning.

We derived metamorphic relations for these subjects based on the guidelines discussed in Section II-C and from some related studies. We illustrate the metamorphic relations chosen using the string searching program. Assume that $X = x_1x_2 \cdots$ is a string, the pattern to search for is pattern $K = k_1k_2 \cdots$, and the output $O(X, K)$ of the matching algorithm is the index, in $X$, of the first matching occurrence if $K$ is found in $X$, and $-1$ if $K$ is not found. We derived the following MRs:

- **MR1. Concatenation**
  \[
  (O(X, K) \neq -1) \Rightarrow O(X \mathbin{\cdot} X', K) \neq -1
  \]

- **MR2. Interleaving**
  \[
  \exists z : (z \not\in X \land X' = x_1z \cdot x_2 \cdots z) \land O(X, K) \neq -1 \Rightarrow O(X', K) = -1
  \]

- **MR3. Reversal**
  \[
  O(X, K) \neq -1 \Rightarrow O(\text{reverse}(X), \text{reverse}(K)) \neq -1
  \]

- **MR4. Character case conversion**
  \[
  O(X, K) \neq -1 \Rightarrow O(\text{toLowerCase}(X), \text{toLowerCase}(K)) \neq -1
  \]

To generate faulty versions of the programs under test, we used the PIT mutation tool\(^2\) for Java and CREAM mutation tool for C\(^3\). Mutation analysis systematically seeds artificial defects into a program based on well-defined rules, generating a set of faulty programs called *mutants*. A mutant is a new version of a program that is created by making a small syntactic change to the original program such as modifying an operator, removing a method call, or negating a condition [5]. For the purpose of analysis, mutation has been shown to be a reasonable substitute for real faults [41].

B. Independent variables

\(a\) Test suites: We generated two sets of test suites for each program: one using DSE; and an extended DSE test suite (EDSE) using metamorphic testing. We used existing guidelines [24] to derive MRs for our subject programs. Based on selected MRs, we generated the extended test suites using automated scripts.

\(b\) Subjects: We tested seven subject libraries from different domains:

- **Sorting algorithms**
  A library written in Java that consists of six different sorting algorithms: bubble sort, heap sort, insertion sort, merge sort, quick sort, and selection sort.

- **String searching algorithms**
  A library of different string searching algorithms (Boyer-Moore algorithm (BM), the Knuth-Morris-Pratt algorithm (KMP), and the Rabin-Karp algorithm (RK)), written in Java.

- **Triangle classifier**
  A well-known triangle classification program, written in Java. It calculates the type of a triangle (scalene, isosceles, equilateral, or not a triangle) based on three input parameters that represent the lengths of the triangle sides.

- **Approximate (fuzzy) string matching**
  A library called Fuzzy String, written in C#. It determines the approximate equality between two strings using 12 different algorithms such as Levenshtein Distance, Longest Common Subsequence, and Sørensen-Dice Distance.

- **Soundex algorithm for phonetic indexing**
  An algorithm written in C# for computing the phonetic soundex code of a given word by collecting its consonants into classes based on the sounding similarity between them.

- **Traffic Collision Avoidance System (TCAS)**
  An algorithm written in C# for computing the phonetic soundex code of a given word by collecting its consonants into classes based on the sounding similarity between them.

<table>
<thead>
<tr>
<th>Subject</th>
<th>#LOC</th>
<th>#Branches</th>
<th>#Methods</th>
<th>#Types</th>
<th>#Mutants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sorting</td>
<td>110</td>
<td>131</td>
<td>6</td>
<td>1</td>
<td>116</td>
</tr>
<tr>
<td>String searching</td>
<td>72</td>
<td>44</td>
<td>3</td>
<td>1</td>
<td>89</td>
</tr>
<tr>
<td>Triangles Classifier</td>
<td>28</td>
<td>16</td>
<td>3</td>
<td>1</td>
<td>18</td>
</tr>
<tr>
<td>Fuzzy matching</td>
<td>389</td>
<td>200</td>
<td>28</td>
<td>4</td>
<td>536</td>
</tr>
<tr>
<td>Soundex</td>
<td>60</td>
<td>38</td>
<td>7</td>
<td>1</td>
<td>149</td>
</tr>
<tr>
<td>TCAS</td>
<td>78</td>
<td>71</td>
<td>10</td>
<td>1</td>
<td>240</td>
</tr>
<tr>
<td>DNA analysis</td>
<td>187</td>
<td>108</td>
<td>16</td>
<td>4</td>
<td>665</td>
</tr>
</tbody>
</table>

Table II summarises subject programs and their relevant properties. LOC refers to lines of code. ‘#Branches’ refers to the number of branches in the source, which is of interest because jCUTE and Pex both aim to achieve full branch coverage on the program. ‘#Types’ refers to the number of types used in the program, which is of interest because most DSE tools can consider only a handful of basic types in their symbolic reasoning.

\(^2\)See pitest.org/
\(^3\)See http://galera.ii.pw.edu.pl/~adr/CREAM/ref.php
In addition, we have used two different tools to generate test suites based on the DSE technique, namely jCUTE and Pex. The purpose is to have at least two representatives of the DSE test generation approach.

As part of the study, we implemented C# copies of many of the Java programs, and found that jCUTE and Pex generated not only similar numbers and types of tests, but generated identical test suites for most programs.

b) Test oracles: We divided our experiment into three categories based on the type of used oracle:

1) No oracle: In this case, a test suite was deemed to have killed a mutant if an unexpected exception was thrown by the subject program during execution. This included exceptions such as null pointers and indexing an array out of bounds.

2) Metamorphic relations: The metamorphic relations were used as a test oracle. Unexpected exceptions were also monitored. This oracle was only applied to the extended test suite, as there are no follow-up test inputs in the original DSE test suite, so there are no metamorphic relations to apply.

3) Golden oracle: The original non-mutated program was used as a test oracle, as well as monitoring for exceptions. While this is not a realistic oracle, in these experiments, the golden oracle is a perfect oracle, replicating ideal behaviour completely, and serves as the optimal result achievable by a test suite.

Using the combination of test oracles and test suites, we end up with five different test suite and oracle combinations, as represented in Table III. The combination of DSE + MR is not sensible, because a metamorphic-relation-based oracle relies on there being metamorphic relations between inputs, and this does not occur in the original DSE test suites. Running the original test suite with metamorphic relations is equivalent to running the original test suite with no oracle; as such, we have not included these in the experiments.

C. Measures

In our experiments, we measured two attributes of each of the five test suites in Table III:

1) Branch coverage: Given that most DSE tools, including jCUTE and Pex, aim to achieve full branch coverage, we used this as a measure of code coverage. We used the EclEmma code coverage tool for Java, and NCover for C#.

2) Mutation score: After generating the mutants as mentioned above, each mutant \( p' \) was executed against a test suite. If the output of executing \( p' \) is different from the output of \( p \) for any test, then the mutant \( p' \) is considered killed by that test, otherwise it is considered alive, and the test suite failed to detect that fault. The mutation score is the ratio of the number of killed mutants over the total number of mutants, and is a proxy measure of the ability to find real faults in programs.

We did not exclude equivalent mutants – those mutants that are syntactically different from, but semantically equivalent to, the original program, and therefore cannot be killed.

V. RESULTS

This section discusses the results from two different perspectives: (1) the effect of the type of test oracle used on the ability to uncover faults; and (2) an analysis of whether the effect of metamorphic testing is based on the size of the extended test suite or the relations themselves.

A. Impact of metamorphic test generation

Figure 2 presents the average branch coverage for the five different test suites in the experiment, and Figure 3 gives the average mutation scores. Table IV gives a per-program breakdown of these. In the table, each tested program has two corresponding rows of data: one for the original DSE test suite, and one for the extended DSE test suite based on the metamorphic relations. The coverage for each test suite is listed, as well as the mutation scores for each type of oracle. Mutation scores for the DNA analysis library are missing due to an issue with the CREAM mutation tool, which we could not get to run on this library.

In this section, we analyse the differences amongst the three types of oracle, comparing the original DSE test suites for no oracle and golden oracle, against the MT-extended test suites.
**c) Code coverage:** A clear observation regarding code coverage is that the original DSE test suites achieve high branch coverage. The extended test suites increase branch coverage in most, but not all, cases. Of course, the extended test suites must cover at least as many branches as the original test suites. Looking at test suite size, we can see that the slight increase of branch coverage is likely related to the size of the test suite.

More detailed analysis supports this. For example, in the triangles classifier program, we found that one of the missed branches belongs to the compound conditional:

\[
\text{if } (\text{Side1} \leq 0 \text{ \&\& } \text{Side2} \leq 0 \text{ \&\& } \text{Side3} \leq 0)
\]

This path represents an invalid triangle where the length of each side is non-positive. However, in the case of using MT, it generated a follow up input 0,-1,-1 from the test input 1,0,0 using the Subtraction property, which exercised the true case for the branch. This demonstrates that the application-specific tests can add value with respect to coverage.

In addition, the code coverage achieved by the extended test suites in Fuzzy string matching program is more than three times the coverage of the original DSE test suites. By inspecting the code, we found that DSE test suites was unable to handle a parameter of type `enum`, and always generated the same value of that parameter. Similarly, the reason of the high coverage achieved by the extended test suites in DNA analysis program in which three of the tested methods have parameters of type Graphics and Image. Pex was unable to generate inputs for those types that are able to exercise different paths in the related code.

**d) Mutation score:** Table IV shows that even when using no oracle, the DSE test suites can detect a reasonably high number of mutants. The only exception is the triangle classification program, for which the mutant score was 0%. This program contains no arrays and throws no exceptions, so this result is unsurprising. In all other cases, using no oracle, the test suites were able to kill more than half as many faults as the golden oracle, which is quite remarkable given that no domain-specific properties are checked.

Using MT as test oracle increases the mutation score by a non-trivial amount in all the subjects. In many cases, the mutation score for the MT oracle is close to that of the golden oracle, and in a handful of cases, exceed the scores achieved by the original DSE suite with a golden oracle.

To get a clearer idea of which type of oracle used with test suites was able to detect more faults, Table V categorises the generated mutants for the TT subject program, and analyses which oracles detected the mutants.

One can see that test suites failed to detect any fault in the program when no oracle was used to check the test output; namely the introduced faults (mutants) will not cause any program crashes or uncaught exceptions, so the program will still produce an output which is incorrect based on its intended functionality. Conversely, using a perfect oracle that checks the correctness of the test output based on the test input improved the test results and increased the mutation score to 82%. Despite this, it detected only two out of the six mutants found by the branch.

Recall that equivalent mutants have not been excluded from our experiments.
in the category that changed the condition. MT as oracle even performed slightly better in detecting mutants generated by changing the conditional boundary compared to the perfect oracle. We attribute this to the increased size of the test suites that are generated based on the MRs.

B. Test suite size vs. oracles

To understand the reason behind the higher effectiveness achieved when MT was used as an oracle, we analyse the effectiveness of the DSE test suites when they are extended by using metamorphic testing. By this, we aim to understand what effect the size of the test suite has on the results. We used the two-way factorial ANOVA model with rank substitution to analyse the critical effects of the two independent variables: test suite and oracle type. For DSE + MR, we use the values from DSE + No oracle, as this is representative of this case.

Table VI presents a factorial experiment table for the mutation scores, which averages the rows and columns of the average data from Table IV. The \( p \) value for all rows was \( p = 0.016 \), while for all columns \( p = 0.0003 \), indicating that the results for the rows and columns are statistically significant. The critical values for a Tukey HSD pairwise test at significance level 0.05 are 15.63 for test suite, and 23.15 for oracles. The differences between the averages of the golden oracle and the other two oracles are therefore significant at the 0.05 level, while the results between no oracle and MR are not significant to the 0.05 level. The results are significant at the 0.05 level for the test suites. Thus, the results indicate that adding metamorphic tests has a significant effect on mutation score, while introducing an oracle has a positive effect, with the golden oracle being significant at the 0.05 level.

C. Threats to validity

There are several threats to validity of this study. The main internal threat to validity is the use of mutation analysis as a proxy for the ability of a test suite to find faults. Although a strong correlation has been shown between mutants and real faults with regards to test suites [41], using real faults could conceivably produce different results.

There are several external threats to validity. First, the sample size of programs is limited, affecting the significance of the results, and second, the program sizes are small compared to programs found in industry. Finally, most of the programs lend themselves to metamorphic testing, as there are some clear and simple metamorphic relations for them. Larger-scale programs especially may not have such effective relations.

D. Discussion

Returning to the questions posed in Section I, we draw some conclusions, based on the results presented above.

First, DSE with no oracle is remarkably good at finding faults, achieving reasonably high mutation scores, even when compared to the golden oracle.

Second, the use of metamorphic testing as a test oracle achieves a higher mutant score than using no oracle, even with the same number of tests, supporting the hypothesis that metamorphic testing can uncover additional faults on top of DSE. This answers question Q1 in the affirmative.

Third, extending test suites with follow-up tests generated by metamorphic testing increased branch coverage. In five of the six programs that did not achieve 100% coverage, metamorphic testing increase the branch coverage score, albeit negligibly in most cases. However, in the one program in which coverage was low (23%), metamorphic testing increased the coverage significantly (to 82%). In all other cases, the branch coverage scores were already high with DSE alone (\( \geq 85\%)\). This answers question Q2 in the affirmative.

Finally, while the DSE approach was able to achieve high code coverage, its overall effectiveness should not be judged based on that criterion alone: the mutation scores increased significantly with the addition of oracles.

Overall, we observe that, when no oracle is available for the tester, DSE can still be an effective approach. However, adding information to the test suite in the form of MRs, as in the case of extended DSE test suites, can increase the value of using DSE. However, if a complete oracle is available, the DSE approach is definitely an effective approach to generate high quality test suites automatically.

VI. Conclusion

In this paper, we assessed whether the use of metamorphic testing principles can increase the value DSE-generated test data, by covering some parts of the code that are left uncovered by DSE and/or through the ability of metamorphic relations to detect domain-specific faults. We undertook a controlled experiment comparing the effectiveness of DSE based on code coverage and mutation score. The results of the study indicate that, while the increase in coverage is limited, it can significantly increase coverage in programs with data types that are difficult to reason about symbolically—in those cases, DSE effectiveness is limited. Further, metamorphic testing has the potential to increase the fault-detecting ability of a DSE-generated test suite considerably.

In future work, we will extend the study to larger-scale programs. In addition, we wish to compare the use of metamorphic testing + DSE against metamorphic testing + random testing, to see if using source tests generated by DSE are more valuable for metamorphic testing than random testing.

ACKNOWLEDGMENTS

The first author acknowledges support from Taibah University through a scholarship to pursue graduate studies at the University of Melbourne.