Generating Source Inputs for Metamorphic Testing Using Dynamic Symbolic Execution

Eman Alatawi
The University of Melbourne
e.alatawi@student.unimelb.edu.au

Tim Miller
The University of Melbourne
tmiller@unimelb.edu.au

Harald Søndergaard
The University of Melbourne
harald@unimelb.edu.au

ABSTRACT

Metamorphic testing uses domain-specific properties about a program's intended behaviour to alleviate the oracle problem. From a given set of *source test inputs*, a set of follow-up test inputs are generated which have some relation to the source inputs, and their outputs are compared to outputs from the source tests, using *metamorphic relations*. We evaluate the use of an automated test input generation technique called *dynamic symbolic execution* (DSE) to generate the source test inputs for metamorphic testing. We investigate whether DSE increases source-code coverage and fault finding effectiveness of metamorphic testing compared to the use of random testing, and whether the use of metamorphic relations as a supportive technique improves the test inputs generated by DSE. Our results show that DSE improves the coverage and fault detection rate of metamorphic testing compared to random testing using significantly smaller test suites, and the use of metamorphic relations increases code coverage of both DSE and random tests considerably, but the improvement in the fault detection rate may be marginal and depends on the used metamorphic relations.

1. INTRODUCTION

Metamorphic testing (MT) is a technique that aims to mitigate the problem of oracle unavailability in software testing by using existing test suites, call *source tests*, to generate *follow-up tests* based on a set of required properties of the targeted functionality of the program. The follow-up tests come about through some systematic perturbations of the source tests, and the expected effect of the perturbations is expressed as *metamorphic relations*, or MRs. When a program is found to violate one of these relations, it signals a fault in the program (assuming the relation is correct) [1]. Effectively the MRs are partial oracles.

For a simple example, an MR for a function intended to produce the absolute value of its input may be

$$m = -n \implies O(m) = O(n) \qquad (1)$$

MET'16, May 14-22, 2016, Austin, TX, USA
© 2016 ACM. ISBN 978-1-4503-4163-9/16/05…$15.00
DOI: http://dx.doi.org/10.1145/2896971.2896980

where $O(x)$ denotes "the output corresponding to input x". In this example the perturbation used is the function that negates its input. The given MR is not only natural but also correct, even in a context of fixed-size integers and twos complement arithmetic. (Other seemingly correct statements, such as $O(n) \geq 0$, fail to hold in this context where the negative numbers outnumber the positive numbers.)

The process by which MT is conducted can be costly and error prone for large programs, owing to the manual work required in extracting MRs, transforming the source tests, and comparing tests outputs [2]. Add to that the challenge of creating source test inputs either manually or randomly (which are the most used techniques with MT). The former technique is difficult and time consuming for non-trivial programs, while the latter can lead to defects remaining undetected in the program under test owing to its randomness.

Although a metamorphic relation's ability to detect faults depends on the nature of the fault, the nature of the relation, and the test inputs [1], the main focus of study has been on how to apply MT in different domains, how to derive MRs, and how to improve test execution. There has been little research on how to generate source tests effectively and automatically, which criteria (size, coverage) to satisfy in source test inputs for an effective MT, and what the best test input generation techniques may be for this.

In this study, we investigate the use of *dynamic symbolic execution* (DSE) to work jointly with MT to generate source test inputs. DSE is usually seen as an alternative technique, applicable for fuzzing only. It uses intelligent exploration of a program's execution paths; the resulting test method is therefore known as "white-box fuzzing". Given a program and an input, DSE uses symbolic execution to record how the input affects the control flow in the program. More precisely, it executes a program in two different ways simultaneously. The DSE tool is seeded with a concrete input, and it traces the execution on this input, while also symbolically executing the same program path. Along the way it collects and stores symbolic constraints at branch points. The constraints collected along a path form a *path constraint* for that path, which explicates (or approximates) more general input conditions that will cause that path to be followed. By selectively negating one or more of these constraints and requesting (from a constraint solver) a concrete solution for the modified constraint, 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 are matched; e.g. branch coverage [3].

Using the popular DSE tool Pex [4], we compare DSE to random testing, to evaluate how each technique can affect MT's test coverage and fault detection ability. In addition, we study the extent to which utilising MRs can automatically improve generated test suites by extending them, based on a program's properties. We pose two questions:

DSE as MT source test seeding: How much will DSE generated source test inputs improve the effectiveness of MT, compared with random inputs?

MT for test enhancement: Can the use of MRs to generate follow-up tests be an effective technique to enhance source test inputs generated by either DSE or random testing?

We have conducted experiments using five small, but non-trivial C# programs. We generated test suites from these using random and DSE automated test input generation techniques, measured their code coverage, and used a 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 perfect, or "golden" test oracle, which is the original, non-mutated version of the program, as an upper baseline.

Our results show that DSE improves test coverage and fault detection of MT compared to random testing, and that MRs increase coverage and fault detection ability of DSE and random testing themselves without incurring high cost.

2. BACKGROUND AND RELATED WORK

Source test cases are used in MT to generate follow-up tests, and to compare their executions against the derived MRs to judge the correctness on the test output in the absence of test oracles. These source tests should run correctly without detecting any faults in the unit under test, and can be generated by any traditional test input generation technique. The quality of source test inputs is a main factor that could affect MT in general, but relatively little attention has been paid to this aspect of MT [5].

In their recent literature review, Segura et al. [6] found that over half of the available studies on MT adopted random testing to generate source test cases, while almost one third used existing test suites, and only few studies used tool–based techniques such as symbolic execution, constraint programming, or search–based testing. Although this demonstrates the ability of MT to be used with different techniques, it is not clear what is required of a source test suite to be able increase the value of MT (e.g., achieved coverage, size), and what the best test input generation techniques may be that can satisfy these requirements.

Gotlieb and Botella [7] use an automatic test data generator, based on constraint programming, to generate source test inputs with high structural coverage. The results show that the MT paradigm can be completely automated. Chen et al. [1] discuss the use of both special values testing and random testing to generate source tests. The goal is to test the critical points of the program. The authors found that the use of random test values is better because of its strong capability to detect more faults, and to cover more of the test domain by providing larger test suites. Wu [8] uses MRs to generate source test inputs using a concept of iterative metamorphic testing, finding that this is able to derive test inputs that are better than those of traditional MT as well as special values testing. Dong et al. [9] utilise program path analysis to improve the generated test inputs in iterative metamorphic testing by covering more paths using fewer test cases. The initial source tests were generated randomly in this study. Batra et al. [10] have used genetic algorithms to generate test inputs to minimise the total number of source tests and to identify the best MRs. Chen et al. [5] propose a technique to reduce the number of test cases needed in MT, based on ad hoc criteria for the division of an input domain into equivalence classes; the aim is to reduce the number of test inputs without compromising the capability of detecting mutants—thus enhancing MT effectiveness.

Dong et al. [11] use symbolic execution for source test input generation. In their approach, a symbolic input is generated for each executable path of the program, and MRs are derived based on the relation between this input and its corresponding output. After that, source test inputs that correspond to the symbolic inputs are generated. This ensures high coverage, supports MR generation and reduces the number of test cases.

The main goal of the studies discussed above is to generate source test suites that are small in size and are able to cover the program under test as thoroughly as possible, in order to increase the fault detection capability of MT. However, the studies use very few automated test generation tools to generate source test inputs.

We are interested in the general question of how/whether we can capitalize on existing program analysis or exploration tools to work in synergy with MT. We have previously utilized MT to improve the effectiveness of DSE generated test inputs by combining DSE's ability to detect generic faults with high code coverage, and MT's ability to detect faults related to functional properties of programs in the absence of a test oracle [12]. We have found that the use of MT increases the fault detection ability of DSE based tests by helping them detect domain specific faults in addition to their original strength in detecting generic faults.

In this paper our focus is on source test input generation with MT, utilising DSE. We are interested in how DSE may enhance MT. In particular, we compare the effectiveness of DSE against the standard approach used in MT: randomly selected source tests. In this context, part of the attractiveness of DSE is its ability to keep test suites relatively small.

3. DSE AS MT SOURCE TEST SEEDING

To understand the relation between source test input generation and MT effectiveness, we can compare DSE generated source tests with tests generated randomly to investigate how that could affect MT effectiveness in terms of both code coverage and fault finding ability. Although random testing can be effective at finding faults, it is also well-known that it provides low code coverage [13]. DSE tends to achieve higher code coverage with fewer tests. Our hypothesis is that DSE based source tests will allow MT to detect more faults, compared to random source tests.

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 inputs. In this case, follow-up tests are guaranteed to cover at least as much as source tests, and that can help detect more code that violates the

MRs and causes program faults.

Listing 1:

```
1 int signalsReachedThreshold (int[] signals) {
2     int count = 0, threshold = 50;
3     for (int i = 0; i < signals.Length; i++) {
4         if (signals[i] == threshold)
5             count--;          // logical error
6     }
7     return count;
8 }
```

Consider a function which, given an input array, returns the number of elements with a particular value (Listing 1). Using random testing to initialise the test input `signals` with n 32-bit signed integers, the "then" branch of the conditional has a vanishingly small probability of being reached. However, using DSE for test input generation we are highly likely to generate input that leads to the line being reached. DSE starts by initialising the input array with some arbitrary concrete values (e.g., $\{0, 0\}$), then executes the function body both concretely and symbolically to collect the path condition along the execution path. Assuming the condition at line 4 is evaluated as false, DSE negates that condition in the next run, generating a new condition to force the execution to reach line 5. Solving the modified path constraint leads to a new concrete input (say, {50,0}) to satisfy the condition. By repeating this process, DSE achieves 100% code coverage for the function under test.

In fact Pex generates a test suite with seven test inputs: null, {}, {0}, {50}, {50,0},{50,50}, {50,0,0} with 100% block coverage. Six of these pass without revealing any fault and we use those six as source test inputs for MT in this example.

To compare the code coverage achieved by Pex and a randomly generated test suite, we generate different test suites with different sizes: size 6 (i.e., the same size as the Pex generated test suite), as well as larger sizes (see Section 5). As expected, random testing does not reach the offending part of the program, even with large test suites—DSE generates fewer tests but achieves higher coverage compared to random testing.

To understand how this may affect MT effectiveness, assume that we do not have an oracle to determine whether the test output is correct, but we use MT to alleviate the oracle problem in this example. Let us assume we have derived the metamorphic relation

$$s' = s{+}{+}s \implies O(s) \leq O(s') \tag{2}$$

We can generate the follow-up tests by duplicating the arrays in the source test suites (both the suite generated by Pex and the suite generated by a random approach). Then we execute the source and the follow-up tests, followed by an evaluation of whether (2) holds or not. We find that 4 out of 6 tests generated by Pex are able to detect the fault by violating the defined MR, whereas the random test suite of same size is unable to detect the fault. In fact, making the random test suite much larger does not help. Our hypothesis is that this phenomenon is common.

4. MT FOR DSE TEST ENHANCEMENT

In this section, we focus on the effectiveness of using of MRs in generating additional tests from a given test suite generated randomly or by using DSE. Without considering the oracle part, which involves evaluating whether the MRs hold between source and follow-up tests in MT, we are interested in the information that MRs could add to the original test suites generated automatically.

Using MRs to extend test suites generated randomly or by DSE has potential to cover additional parts of the code and detect more faults than either random or DSE based test suites. We will refer to this process of generating additional tests using passed source test inputs and according to a set of derived MRs as *"metamorphic test generation"*.

Random testing, for example, depends on probabilities only, and then has low chances in detecting the faults that can only be discovered by a small set of the input domain [13]. DSE is able to achieve high code coverage with small sets of tests, assuming it does not fall victim to inherent limitations such as inability to handle complex (non-linear) constraints, complex loops and code that causes path explosion. These limitations can result in missing interesting blocks of the code that could have critical faults.

Using MRs can add some useful information about program properties to the additional tests. The new tests can be used to enhance automatically generated test suites and guide them to cover more interesting parts of the code, and then detect more faults. Consider this simple function, intended to produce the absolute value of its argument:

Listing 2:

```
1 int abs(int num) {
2     if (num > 0)
3         return num;
4     return num; // '-' inadvertently omitted
5 }
```

Using Pex [4] with its standard settings, we produce the following test set for `abs`: $\{0, 1\}$. This satisfies coverage criteria in that it yields 100% block coverage. However, even in the presence of a perfect oracle, the error in line 4 remains undetected. Using the DSE generated tests (0 and 1) metamorphic testing can immediately make up for this.

Now, based on the number of tests generated by Pex (2), we generated three test suites of the sizes: 2, 20, and 200. The coverage achieved by these test suites after five runs is 75% for each. By the randomness of this technique, the tests are either positive or negative without realizing the need for a test suite that contains values on either side of 0, to cover all "interesting" cases.

Assume we have a complete oracle that checks the correctness of the function's output, and we also have the MR (1) from Section 1. The original tests include 0 or positive values as test inputs for which the faulty `abs` function can produce a correct answer. Clearly, although DSE generated tests achieve complete coverage, they cannot reveal the error at line 4, because the value 0 was picked by Pex to satisfy the condition `num` ≤ 0.

However, with the use of the MR (1), we generate additional tests by transforming the original test inputs, that run successfully without revealing any faults, based to the derived MR (i.e. by multiplying the source tests by -1), which is similar to the process of generating follow-up tests in MT.

By executing these additional tests and evaluating their output using our complete oracle, we can see that the function will produce negative output values which are not as expected by the oracle. In this case, the additional tests reveal the fault in the function. This is done by giving an interesting input for DSE based test suite other than 0 to exercise the corresponding path, and also by giving two additional tests for random based test suites that help to achieve 100 coverage and in the same time detect the fault.

Using MRs can be an efficient and cost effective supportive technique for automated test input generation. Except for identifying MRs, the process can be fully automated [14].

5. EXPERIMENT DESIGN

We now explain the design of our experiment, which aims to answer the research questions outlined in Section 1. To answer the first question (effectiveness of DSE to generate source tests) we compare coverage and mutation score achieved by test suites generated automatically by random and DSE, as they are used as source tests with MT. To answer the second question (effectiveness of MR of follow-up tests for DSE test suites) we compare coverage and mutation score of the used test suites before and after augmenting MRs and with/without oracles.

We have used five well-known small, but non-trivial programs, and generated tests from these using random testing as well as the DSE tool Pex.

Then we extended the resulting DSE and random test suites using a set of simple metamorphic relations specific to the programs (see the appendix). We measured the coverage of the original and extended test suites, and measured the mutation score of these given four different cases: (1) source tests with no oracle; (2) extended MR tests with no oracle; (3) source tests with MT as an oracle; and (4) source tests with a perfect oracle, namely the original non-mutated program, which serves as a theoretical upper bound for test oracles.

5.1 Objects of Analysis

The five subject programs are written in C# and come from different domains. They implement the Soundex algorithm for phonetic indexing, and edit distance algorithm, approximate string matching, a traffic collision avoidance system (TCAS), and a Big Integers arithmetic library. Table 1 summarises the subject programs and their relevant properties: LOC (lines of code), number of test inputs generated by Pex, number of derived MRs, and number of mutants generated for the mutation analysis. The appendix gives more detail.

To generate faulty versions of the programs under test, we used the CREAM mutation creator tool for C#[1]. 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 [15]. For the purpose of analysis, mutation has been shown to be a reasonable substitute for real faults and it is widely used in metamorphic testing related studies [6].

5.2 Independent Variables

Table 1: Subject programs used in the experiment

Subject	#LOC	#DSE-Source Tests	#MRs	#Mutants
Soundex (S)	60	21	5	41
Edit distance (E)	15	5	2	104
Approx. string matching (A)	48	12	3	36
TCAS (T)	78	11	4	231
BigInt (B)	201	5	3	995

Test suites.

We generate two types of test suites for each program based on the used test generation approach: Random based test suites, and DSE based test suites. In fact we create four different Random based test suites of increasing size for a more reliable analysis—different runs of random testing gives different results in terms of coverage and mutation analysis, and this way we get an indication of the degree to which randomness affects the outcome. The first random test suites is created with same size n as the DSE-based test suite, and the remaining three random test suites are created with sizes $10n$, $100n$, and $1000n$. Thus, we have altogether five different test suites that all run successfully on the program under test without revealing any faults. We refer to these tests suites as **B**, for *Base source tests*.

We then extend these test suites by generating follow-up tests by transforming each test input based on the derived MRs for each subject. We refer to these test suites as **M**, for *Metamorphically generated tests*. We use M to understand how effective they are (being generated from a specific base source test suite to help MT to detect more faults to alleviate the absence of the test oracle, and to judge their effectiveness as standalone additional tests without considering MT as an oracle, in which we evaluate their fault finding effectiveness in detecting generic faults (i.e. without an oracle), and also when they are combined with a perfect test oracle.

Test oracles.

We divided our experiment into three categories based on the type of oracle used:

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; e.g., null pointers and indexing an array out of bounds.

2. *Metamorphic relations*: The metamorphic relations were used as a test oracle (i.e. represents performing MT). Unexpected exceptions were also monitored.

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, it servers as the optimal result achievable by a test suite.

Measures.

We measured two attributes of each of the five test suites as follows:

1. *Block coverage*: We used the code coverage tool included in VS.net 2015[2]

2. *Mutation score*: After generating the mutants as mentioned above, each mutant was executed against the

[1]See http://galera.ii.pw.edu.pl/~adr/CREAM/ref.php

[2]https://msdn.microsoft.com/en-us/library/dd537628.aspx

Table 2: Experiment Results

		Coverage %		Mutation score %				
Subj	**ST**	**B**	**M**	**MT**	**BN**	**MN**	**BP**	**MP**
S	DSE	100	100	49	20	20	83	95
	R_n	90	92	24	22	24	56	76
	R_{10n}	92	95	27	22	24	56	76
	R_{100n}	92	93	29	24	24	59	78
	R_{1000n}	96	100	39	24	24	76	83
E	DSE	100	100	77	37	37	71	100
	R_n	94	100	50	35	35	51	51
	R_{10n}	94	100	50	35	35	51	51
	R_{100n}	94	100	50	35	35	53	61
	R_{1000n}	94	100	52	35	35	60	61
A	DSE	45	99	47	29	28	53	61
	R_n	41	99	33	0	28	0	64
	R_{10n}	42	99	33	0	28	0	64
	R_{100n}	34	99	33	0	28	0	64
	R_{1000n}	47	99	33	3	28	4	64
T	DSE	97	97	23	5	23	42	44
	R_n	20	21	1	0	1	3	3
	R_{10n}	20	21	1	0	1	3	3
	R_{100n}	28	21	9	5	8	7	7
	R_{1000n}	36	58	9	6	9	8	10
B	DSE	43	59	17	9	16	17	20
	R_n	59	65	20	15	17	17	19
	R_{10n}	59	65	20	15	17	17	19
	R_{100n}	59	65	20	15	17	17	19
	R_{1000n}	59	65	20	15	17	17	19
Avg	DSE	77	91	39	12	16	37	39
	R_n	61	75	22	11	12	21	25
	R_{10n}	61	76	22	11	12	21	25
	R_{100n}	62	77	24	13	13	24	28
	R_{1000n}	67	84	27	13	14	27	29

ST: source tests suite's type (DSE vs random). **B**: base (original) test suite. **M**: metamorphically extended test suite. **MT**: metamorphic testing using **B** and **M**. **BN**, **MN**: **B** and **M** with no oracle, and **BP**, **MP**: **B** and **M** with perfect oracle

given test suite, and was considered *killed* by that test suite if at least one test failed. The *mutation score* is the ratio of the number of killed mutants over the total number of mutants.

6. RESULTS AND DISCUSSION

Table 2 gives detailed results of our experiment. Each tested program has five corresponding rows of data: one for the original DSE test suite of size n, and four for random test suites of size n, $10n$, $100n$, and $1000n$, denoted R_n, R_{10n},R_{100n}, and R_{1000n}. The coverage for each test suite is listed considering two cases: using the base source test suite alone (**B**), and using the extended test suite based on MRs (**M**). The mutation scores are given for each analysed case: (**MT**) as oracle using both **B** and the follow-up tests in **M** (in which the output of **B** and **M** is checked against the defined MRs), base test suites with no oracle (**BN**), metamorphically extended test suites with no oracle (**MN**), **B** with perfect oracle (**BP**), and **M** with perfect oracle (**MP**). All results corresponding to the random test suites are based on the average of 5 runs of the experiment, as the random test suites results can vary among different executions.

Code coverage.

Original coverage of the source test suites generated using DSE is considerably higher than the randomly generated ones of the either same size, 10, 100, or 1000 times as the DSE test suite's size with an average increase of 16%,16%,14%, and 10% respectively. This shows that the source tests generated using DSE are able to achieve higher coverage, compared to the random approach with considerably less number of test inputs. The only exception is in the subject BigInt (B) where random test suites achieve better coverage than DSE generated ones because of the complex object inputs required by the tested functions that Pex could not handle. The results clearly indicate an increase in the code coverage up to 15% on average in all subjects when MRs are used to generate new tests from the original DSE and Random source test suites (B). Looking to the source test suites individually, we can see that the increase of the coverage ranges from 0-54% with DSE based test suites, and 1-58% in random test suites. In fact, the improved coverage is limited when the base DSE and random test suites are able to achieve reasonably good coverage on their own. However, additional tests generated using MRs do improve the coverage significantly when the base source tests struggle to do that. The only exception is in TCAS where random test suites achieve low coverage, and extended test suites add a limited coverage increase of only 3%. This is because the program takes 12 different input parameters, and the MRs used do not help in transforming all the inputs in a way that could lead to diverse program executions.

Looking at test suite size, we can see that the increase of branch coverage in **M** is not strongly correlated with the size of the test suite because extending DSE generated test suites of size n with MRs achieves the best coverage (average of 91%) compared with random test suites of size $1000n$ (average of 84%) for example.

Mutation score.

Source tests and MT. Table 2 shows that using DSE to generate source test inputs for MT increases the fault detection ability of MT in all but one subject with an average of 39% in the mutation score, compared to 22% for a random source test set of the same size n. Growing the random set yields 22%, 24%, and 27% average mutation scores, as the size of the set of random source tests grows to size $10n$, $100n$, and $1000n$. The exception is the BigInt case where randomly generated source tests performed marginally better than DSE. Recall that Pex was unable to achieve high coverage for this subject.

In short, DSE provides effective source tests with a small number of test inputs that achieve high code coverage, and using such source test inputs improves the effectiveness of MT compared to random test input generation in terms of both coverage and fault detection rate. The improvement can be seen clearly in cases where the DSE generated test inputs are able to achieve higher code coverage. However, DSE could perform slightly less than random testing (e.g. in BigInt subject) in the presence of the limitations that could hinder DSE from achieving a reasonable coverage, but it is still an effective automated technique to be used with MT.

DSE as MT source test seeding: Using DSE generated source test inputs improves the effectiveness of MT compared to randomly generated source test inputs in terms of both coverage and fault detection rate.

B vs M. Looking at the difference between base source test suites generated using either DSE or randomly, and the others extended using metamorphic generation (M), we see that, generally, using MRs to extend the original test suites improves the fault finding effectiveness slightly, as indicated by the mutation scores. For example, the original DSE source tests have 12% mutation score on average when used without any oracle, as compared to 16% on average when MRs are used. The improvement—albeit small—remains present even when a perfect oracle is used to evaluate the correctness of test outputs. This shows that MRs can add useful test inputs to automatically generated test inputs.

In addition, we found that the MR-T increased the mutation score with up to 15% in Soundex subject, and that could be a result of the strength of the used MRs because we use 5 different MRs that lead to substantial transformation of the source test inputs to generated the additional tests.

Notice that when metamorphically extended test suites improved the coverage significantly (such as in approximate string matching (A)), it had a direct effect on the mutation score, increasing it significantly, especially when the perfect oracle is available. This is because MRs used in this subject guided the testing process through different executions, and transformed the source test inputs systematically to cover all possible values for two parameters of type *enum*. Pex, for instance, always generates the same value of that parameter with value 1, and similarly using the random approach could not cover all possible values for these parameters.

In short, using metamorphic test input generation to extend a test suite generated automatically using DSE or random testing increased the code coverage considerably in all subjects and for all source test suites with an average increase of 15%, compared to the original source tests. However, the increase in the fault detection rate is slight with an average of 3% only in the presence of a perfect test oracle.

MT for test enhancement: Using metamorphic test input generation to extend a test suite generated automatically using DSE or random testing can increase their code coverage considerably, but the improvement in the fault detection rate can be marginal and depends mainly on the strength of the used MRs.

Threats to validity.

The investigation that we have conducted is preliminary; the code base that we have used is not large enough to allow a meaningful statistical analysis. These are external threats to validity—the sample size of programs is limited, and the program sizes are small compared to programs found in industry.

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 [16], using real faults could conceivably produce different results. In addition, the use of random testing can produce different results in terms of coverage and mutation score each run, but we consider the average results after repeating the experiments 5 times with randomly generated tests suites.

7. CONCLUSION

In this paper, we assessed whether the use of DSE to generate source test inputs for MT can increase the value of this approach by extending its ability to detect faults, and whether the use of metamorphic test inputs generation can be effective to enhance automatically generated tests suites using DSE or random testing by adding better tests. We undertook a controlled experiment comparing the effectiveness- in terms of code coverage and mutation score- of DSE based source test inputs and randomly generated ones when they are used with MT. The results of the study indicate that, using DSE generated source test inputs improve the effectiveness of MT compared to random test inputs in terms of both coverage and fault detection rate. In addition, the use metamorphic relations to extend a test suite generated automatically using DSE or random testing can increase their code coverage considerably, but the improvement in the fault detection rate can be marginal and depends mainly on the strength of the MRs used.

Acknowledgments

The first author gratefully acknowledges support from Taibah University through a scholarship.

8. REFERENCES

[1] T. Y. Chen, F.-C. Kuo, Y. Liu, and A. Tang, “Metamorphic testing and testing with special values,” in *Proc. 5th Int. Conf. Software Eng., Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*. Int. Assoc. Computer and Information Science, 2004, pp. 128–134.

[2] C. Murphy, K. Shen, and G. Kaiser, “Automatic system testing of programs without test oracles,” in *Proc. 18th Int. Symp. Software Testing and Analysis (ISSTA’09)*. ACM, 2009, pp. 189–200.

[3] P. Godefroid, M. Levin, and D. Molnar, “Automated whitebox fuzz testing,” in *Proc. Network and Distributed System Security Symp. (NDSS)*. The Internet Society, 2008, pp. 151–166.

[4] N. Tillmann and J. De Halleux, “Pex—white box test generation for .NET,” in *Tests and Proofs*, ser. LNCS. Springer, 2008, vol. 4966, pp. 134–153.

[5] L. Chen, L. Cai, J. Liu, Z. Liu, S. Wei, and P. Liu, “An optimized method for generating cases of metamorphic testing,” in *Proc. 6th Int. Conf. New Trends in Information Science and Service Science and Data Mining (ISSDM)*. IEEE, 2012, pp. 439–443.

[6] S. Segura, A. B. Sánchez, and A. Ruiz-Cortés, “Metamorphic testing: A literature review,” University of Seville, Spain, Tech. Rep. ISA-15-TR-01, 2015.

[7] A. Gotlieb and B. Botella, “Automated metamorphic testing,” in *Proc. 27th Ann. Int. Computer Software and Applications Conf. (COMPSAC)*. IEEE, 2003, pp. 34–40.

[8] P. Wu, “Iterative metamorphic testing,” in *Proc. 29th Ann. Int. Computer Software and Applications Conf. (COMPSAC)*. IEEE, 2005, pp. 19–24.

[9] G. Dong, C. Nie, B. Xu, and L. Wang, “An effective iterative metamorphic testing algorithm based on program path analysis,” in *Proc. 7th Int. Conf. Quality Software (QSIC’07)*. IEEE, 2007, pp. 292–297.

[10] G. Batra and J. Sengupta, “An efficient metamorphic testing technique using genetic algorithm,” in *Information Intelligence, Systems, Technology and Management*. Springer, 2011, pp. 180–188.

[11] G. Dong, T. Guo, and P. Zhang, “Security assurance with program path analysis and metamorphic testing,” in *Proc. 4th IEEE Int. Conf. Software Eng. and Service Science (ICSESS)*. IEEE, 2013, pp. 193–197.

[12] E. Alatawi, T. Miller, and H. Søndergaard, “Using metamorphic testing to improve dynamic symbolic execution,” in *Proc. 24th Australasian Software Eng. Conf. (ASWEC 2015)*. IEEE, 2015, pp. 38–47.

[13] P. Godefroid, N. Klarlund, and K. Sen, “DART: Directed automated random testing,” in *Proc. ACM SIGPLAN Conf. Programming Language Design and Implementation (PLDI’05)*. ACM, 2005, pp. 213–223.

[14] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou, “Case studies on the selection of useful relations in metamorphic testing,” in *Proc. 4th Ibero-American Symp. Software Eng. and Knowledge Eng. (JIISIC)*, 2004, pp. 569–583.

[15] L. Inozemtseva and R. Holmes, “Coverage is not strongly correlated with test suite effectiveness,” in *Proc. 36th Int. Conf. Software Eng. (ICSE’14)*. ACM, 2014, pp. 435–445.

[16] R. Just, D. Jalali, L. Inozemtseva, M. D. Ernst, R. Holmes, and G. Fraser, “Are mutants a valid substitute for real faults in software testing?” in *Proc. 22nd ACM SIGSOFT Int. Symp. Foundations of Software Eng. (FSE’14)*. ACM, 2014, pp. 654–665.

[17] M. Asrafi, H. Liu, and F.-C. Kuo, “On testing effectiveness of metamorphic relations: A case study,” in *Proc. 5th Int. Conf. Secure Software Integration and Reliability Improvement (SSIRI)*. IEEE, 2011, pp. 147–156.

Appendix

Subject Programs and Metamorphic Relations Used

- **Soundex algorithm for phonetic indexing (SNDX)**
  An algorithm for computing the phonetic soundex code of a given word by collecting its consonants into classes based on the sounding similarity between them. Assume that $X = \{x1, x2, \cdots\}$ is the given word, and the output $O(X)$ will the soundex code of X. We derived the following MRs:
  - MR1. Inclusion
    $z \in \{a, e, i, o\} \Rightarrow O(x_1x_2\cdots) = O(x_1zx_2z\cdots z)$
  - MR2. Duplication
    $O(x_1x_2\cdots x_n) = O(x_1x_2\cdots x_nx_n)$
  - MR3. Reversal
    $|X| > 4 \wedge X \neq reverse(X) \Rightarrow$
    $O(X) \neq O(reverse(X))$
  - MR4. Deletion
    $|X| > 4 \wedge X[n-1] \in \{a, e, i, o\} \Rightarrow$
    $O(X) = O(X - X[n-1])$
  - MR5. Character case conversion
    $O(X) = O(toLower(X))$
- **Edit distance algorithm**
  An implementation of Edit distance algorithm. It calculates the total minimum number of edit operations that transforms one given string x into another one y. We use the following MRs:
  - MR1. Limit case
    $O(X,' \ ') = O(' \ ', X) = |X|$
  - MR2. Concatenation
    $X' = X{+}{+}R \wedge Y' = Y{+}{+}S \Rightarrow$
    $O(X', Y') = min(O(X, Y), O(X', Y) + 1, O(X, Y') + 1)$
    where R and S are random strings of length 1.
- **Approximate string matching**
  A library, i“Fuzzy String”, that determines the approximate equality between two strings using 12 different algorithms such as Longest Common Subsequence, and Sørensen-Dice Distance. Assume that $X = x_1x_2\cdots$ is the source string to be matched, $Y = y_1y_2\cdots$ is the target string to be matched with, $t \in \{strong, weak, normal\}$ is the tolerance level, op is the algorithm used, and output $O(X, Y, t, op)$ is true if and only if the two strings are approximately matched according to tolerance level t. We use the following MRs:
  - MR1. Reversal
    $X' = reverse(X) \& Y' = reverse(Y) \Rightarrow$
    $O(X, Y, t, op) = O(X', Y', t, op)$
  - MR2. Limit cases
    $t \neq weak \Rightarrow$
    $O(X,' \ ', t, op) = false \wedge O(X, X, t, op) = true$
  - MR3. Monotonicity
    $O(X, Y, strong, op) = true \Rightarrow$
    $O(X, Y, normal, op) = O(X, Y, weak, op) = true$
- **Traffic Collision Avoidance System (TCAS)**
  A small aircraft conflict avoidance system. It takes 12 parameters and calculates whether there will be a conflict between a current aircraft and an approaching aircraft. Possible outputs include 0 (no action), 1 (upward), or 2 (downward). We adopted the following MRs from [17]:
  - MR1. Changing parameters randomly
    By randomly changing the parameters values, and cvs = current vertical separation between two aircrafts at the closest point is set to be greater than 600, the following relations should hold:
    $cvs > 600 \wedge O = 0 \Rightarrow O' \in \{0, 1, 2\}$
    $cvs > 600 \wedge O \neq 0 \Rightarrow O' \neq O$
  - MR2. Changing aircraft’s intention
    For otherwise fixed parameters, let O be the output given $aircraft\_intent = true$ and let O' be the output given $aircraft\_intent = false$. Then $O' = O$.
  - MR3. Changing the status of reports validity
    For otherwise fixed parameters, let O be the output given $reports\_valid = true$ and let O' be the output given $reports\_valid = false$. Then $O' = O$.
  - MR4: Changing TCAS confidence indicator
    For otherwise fixed parameters, let O be the output given $TCAS\_has\_high\_confidence = true$ and let O' be the output given $TCAS\_has\_high\_confidence = false$. Then $O' = O$.
- **Big Integers**
  An implementation of BigInteger class that supports large integer arithmetic operations that are not supported natively by the compilers. We focus on the addition and multiplication operations. We use the following MRs for addition (and similar ones for multiplication):
  - MR1. Commutation
    $O(x, y) = O(y, x)$
  - MR2. Association
    $O(O(x + y), z) = O(x, O(y + z))$
  - MR3. Distribution
    $z * O(x, y) = O((x * z), (y * z))$