Case Studies on the Selection of Useful Relations in Metamorphic Testing [⋆]

T. Y. Chen[1], D. H. Huang[1], T. H. Tse[2], and Zhi Quan Zhou[1] [⋆⋆]

[1] School of Information Technology, Swinburne University of Technology,
Hawthorn, Victoria 3122, Australia
Email: {tchen, dhuang, zhzhou}@it.swin.edu.au

[2] Department of Computer Science, The University of Hong Kong,
Pokfulam, Hong Kong
Email: thtse@hku.hk

Abstract. An *oracle* is a mechanism against which the tester can decide whether the outputs of the program for the executed test cases are correct. A fundamental problem of software testing is that, in many situations, the oracle is not available or too difficult to apply. A *metamorphic testing* (MT) method has been proposed to alleviate the oracle problem. MT is an automated testing method that employs expected properties of the target functions to test programs without human involvement. These properties are called *metamorphic relations* (MR). For a given problem, usually more than one MR can be identified. It is therefore interesting and very useful for practitioners to know how to select effective MRs that are good at detecting program defects. This article proposes a guideline for the selection of good MRs for automated testing. The effectiveness of our strategy has been investigated through case studies.

Keywords: Verification and validation, software testing, automated testing, metamorphic testing, metamorphic relation, test oracle, follow-up test case

1 Introduction

1.1 The Oracle Problem

Program correctness has always been a critical issue for both researchers and practitioners. The past decades have shown that the use of formal verification (i.e., program proving) to real-life applications has been very limited [1] due to the difficulties in proofs and automation. Program testing, therefore, remains the most popular means adopted by practitioners [1, 2]. Nevertheless, testing has two fundamental limitations. First, the use of test cases cannot guarantee program correctness on untested inputs [2, 3]. In other words, testing cannot prove the absence of faults in most situations. Secondly, in some situations, it is impossible or practically too difficult to decide whether the program outputs on test cases are correct. This is known as the *oracle problem* [4].

[⋆] This research is supported in part by a discovery grant of the Australian Research Council (Project No. DP 0345147), a grant of the Research Grants Council of Hong Kong (Project No. HKU 7145/04E), and a grant of the University of Hong Kong.

[⋆⋆] Corresponding author.

This article is concerned with the oracle problem. Let p be a program implementing a specification f. Let D represent the input domain. Usually, it is impossible to do exhaustive testing to check whether $p(t) = f(t)\ \forall t \in D$. As a result, a great amount of research in the literature of software testing has been devoted to the development of *test case selection strategies*, aiming at selecting those test cases that have a higher chance of detecting a *failure*. Let $T = \{t_1,\ t_2,\ \ldots,\ t_n\} \subset D$ be the set of test cases generated according to some test case selection strategy, where $n \geq 1$. Running the program on these test cases, the tester will check the outputs $p(t_1),\ p(t_2),\ \ldots,\ p(t_n)$ against the expected results $f(t_1),\ f(t_2),\ \ldots,\ f(t_n)$, respectively. If it is found that $p(t_i) \neq f(t_i)$ for some i, where $1 \leq i \leq n$, then we say a "failure" is revealed and t_i is a *failure-causing* input. Otherwise t_i is a *successful test case*. The procedure through which the tester can decide whether $p(t_i) = f(t_i)$ is called an *oracle* [4]. For instance, let $f(x,\ y) = x \times y$, the test case t_i be $\{x = 3.2,\ y = 4.5\}$, and $p(t_i) = 14.4$. The tester can verify this output either by manually calculating the product of 3.2×4.5 or using the inverse function to check whether $14.4/4.5 = 3.2$, where the inverse can be done either manually or using a correct division program if available. In many situations, however, the oracle is not so easy to apply. In cryptosystems, for example, the operands in the multiplication are multiprecision integers with hundreds of hexadecimal digits. As a result, the output is so large that it is practically too expensive to verify the result. Although the use of the inverse function can help, correct programs for these inverse functions may not be available, and many functions do not have an inverse, such as the Greatest Common Divisor. In this situation, where the oracle is not available or too difficult to apply, there is an "oracle problem" [4]. Other examples include, to name a few, testing programs conducting numerical integrations or solving partial differential equations; deciding the equivalence between the source code and object code when testing compilers; testing programs that calculate combinatorial problems, perform simulations, draw complicated graphics, etc. In fact, even when the oracle is available, if it cannot be automated, the manual predictions and comparisons of the outputs are often expensive and error-prone [5, 6]. As pointed out in [6], the oracle problem has been "one of the most difficult tasks in software testing" but it is often ignored by researchers in software testing.

It should be noted that even when the oracle is not available, the tester usually is somehow still able to check the outcome of the programs to some extent. In [4], Weyuker investigated various approaches to testing programs when there is an oracle problem. Special or simple test cases are often used in such a situation. When testing the sine function, for instance, the special inputs 0, $\pi/4$, $\pi/2$, etc., are standard test cases. Nevertheless, special values cannot give us enough confidence in the correctness of the program on more complex or random inputs. Another practical approach is to check the outputs against properties of the target function known from theory. When testing a program p supposedly implementing the sine function, for example, let 1.28 be a test case. Although we do not know the exact value of $\sin 1.28$, a failure can still be identified if the output $p(1.28) > 1$ because $|\sin x| \leq 1\ \forall x \in R$. By employing more mathematical properties of the sine curve, the range of plausible values of $p(1.28)$ can be further narrowed down greatly.

1.2 Successful Test Cases

No matter how the outputs are checked, with or without an oracle, we all know that in practice most test cases are "successful test cases" (i.e., they do not reveal any failure) if the program is written by a competent programmer [7]. On the other hand, successful test cases have been considered useless in conventional testing because they do not reveal any failure [8]. As a result, in conventional testing the successful test cases have been discarded or retained merely for regression testing later.

Our perspective, however, is different from the conventional view. We argue that successful test cases are informative and should be exploited further in a systematic and cost-effective way. Our argument is based on two observations: First, successful test cases do carry useful information that has been ignored in conventional testing. Fault-based testing [9], for example, is an important breakthrough because it uses successful test cases to prove the absence of certain types of error. Unfortunately, not all testing methods are fault-based and the majority of test cases are successful.

Secondly, no matter whether an oracle is available, testing is expensive in most situations [1] because test case design, implementation, output prediction and comparison, as well as documentation, are labor intensive. Hence, *each test case is valuable*. It is a great waste if most of the test cases are immediately cast off after running once. It is therefore highly desirable to develop methodologies to effectively utilize the successful test cases so that the program can be verified in a more cost-effective way.

1.3 Metamorphic Testing

A *metamorphic testing* (MT) method has been proposed by Chen et al. [10] and further developed ([11–13], among others). It is an automated approach to alleviating the oracle problem and employing successful test cases.

MT is to be used in conjunction with other test case selection strategies. To test program p implementing function f on domain D, let S be the test case selection strategy adopted by the tester, such as branch coverage testing, data flow testing, or just random testing. Let $T = \{t_1,\ t_2,\ \ldots,\ t_n\}\ \subset\ D$ be the test set generated according to S, where $n \geq 1$. If the outputs $p(t_1),\ p(t_2),\ \ldots,\ p(t_n)$ reveal no failure, then we encounter a set of successful test cases.

At this stage, MT can be employed to make use of the successful test cases: By referring to certain properties called *metamorphic relations* (MR) of the target function f, follow-up test cases can be automatically constructed, executed, and checked to further verify program p without the need of an oracle. A "metamorphic relation" is any relation among the inputs and the outcomes of *multiple* executions of the target program. For example, let $p(a,\ b,\ g)$ be a program supposedly computing the numerical integration $\int_a^b g(x)\,\mathrm{d}x$. When $g(x)$ is complicated, there is no straightforward oracle to test the program. Nevertheless, we can identify metamorphic relations known from theory, such as $\int_a^b k \times g(x)\,\mathrm{d}x = k \times \int_a^b g(x)\,\mathrm{d}x$, where k is any constant. Metamorphic testing (MT) checks programs against metamorphic relations (MR). For our example, if the initial test case is $t : \{a = 2.3,\ b = 4.5,\ g = g_1(x)\}$ and no failure is detected, then MT proposes to go one step further to generate one (or more) follow-up test case $t' : \{a = 2.3,\ b = 4.5,\ g = g_2(x)\}$, where $g_2(x) = k \times g_1(x)$ for some constant k, and run the program again on t'. The outputs are then compared against the prescribed MR. If $p(t') \neq k \times p(t)$,[3] then the program must be at fault. Certainly, an MR is a necessary property, but may not be sufficient for program correctness. This is indeed the limitation of all testing methods.

Since it is the relation among multiple executions rather than the correctness of individual outputs that is checked, MT is performed regardless of the existence of an oracle. In addition, because the whole process can be fully automated without human involvement, MT is an easy and efficient approach to exploiting successful test cases.

In fact, the idea of employing identity relations to check programs is not new. In [14], for example, many identity relations are used to test programs, such as testing program $p(x)$ against the identity "$e^a \times e^{-a} = 1$", where the target function is e^x. Identity relations are also intensively used in fault-tolerance techniques [15], *program checker* [16, 17] and *self-testing/correcting* [18], and so on. There are, however, great differences between these methods and metamorphic testing. First, MT is to be used in conjunction with a test case selection strategy S, where S can be any black- or white-box testing strategy. A test set T generated from S must also exist in the first place. If no failure can be revealed by T, then MT can be applied to generate a follow-up set of test cases to accompany T to further verify the program against selected metamorphic relations, which are necessary properties for program correctness. Secondly, an MR is not necessarily an identity relation. Any relation involving two or more executions of the target program is an MR. To name a few, it includes inequalities, convergence properties, subsumption relations in set theory, and so on. In [11], for example, we employed the convergence property as an MR to test the program solving the partial differential equation.

We have found that metamorphic relations can be identified in a wide range of applications. In fact, for most problems, more than one MR can be identified. Let us take the numerical integration program as an example. Apart from the property already discussed, the following properties can be identified as MRs as well: $\int_a^b \big(g_1(x) + g_2(x)\big)\,\mathrm{d}x = \int_a^b g_1(x)\,\mathrm{d}x \ + \ \int_a^b g_2(x)\,\mathrm{d}x$, $\int_a^b g(x)\,\mathrm{d}x = -\int_b^a g(x)\,\mathrm{d}x$, $\ldots$ In fact, even for a given property like $\int_a^b k \times g(x)\,\mathrm{d}x = k \times \int_a^b g(x)\,\mathrm{d}x$, different valuations of k can be regarded as different MRs. It is, therefore, very useful and important to know how to select good metamorphic relations that have a higher chance of revealing failures in testing.

This article proposes a guideline for selecting good metamorphic relations for software testing. In Section 2, we shall conduct case studies to investigate how likely it is to select good MRs solely based on theoretical knowledge of the problem domain. Our result shows that theoretically stronger MRs do not necessarily have a higher failure-detecting capability. As a result, the program structure must be considered for the selection of good MRs. In Section 3, we propose our MR selection strategy based on the white-box knowledge of the program structure. Our experiment result shows that the proposed method is effective. Section 4 will conclude the paper.

[3] In practice, some rounding error will be allowed due to floating-point arithmetic.

2 Comparing MRs from a Black-Box Perspective

Since MRs are identified with regard to the original specification rather than the program under test, it would be ideal if we could also have a way to select good MRs without white-box knowledge of the program. In this section, therefore, we shall treat the program as a black box and discuss the selection of MRs solely based on the specification, i.e., the target function.

For a given specification, suppose two MRs have been identified, namely R_1 and R_2. An intuition is that if R_1 is stronger than R_2 from theory, i.e., if $R_1 \Rightarrow R_2$, then R_1 is likely to have a higher chance of detecting failures. However, since the implementation is not necessarily correct, R_1 is not necessarily better than R_2 in revealing the defect in the implementation. Nevertheless, in order to provide a practical guideline for software testers, it is still worthwhile to investigate how likely the stronger relations are better than the weaker ones for testing and the reasons behind.

2.1 A Case Study on the Shortest Path Program

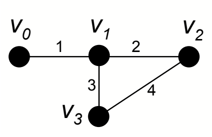
Our first case study is on a program $ShortestPath(G,\ a,\ b)$ that implements Dijkstra's algorithm to find the *shortest path* between vertices *a* and *b* in graph *G* and also output its length, where *G* is an undirected graph with positive edge weights. When *G* is nontrivial, the program is difficult to test because no oracle can be practically applied. Nevertheless, many MRs can be identified for this problem, with which MT can be performed.

Left Circular Shifts as the MRs A property that can be commonly found for programs in graph theory is the permutation property. Let $(G_1,\ a_1,\ b_1)$ be the first input to program *ShortestPath*. Let $(G_2,\ a_2,\ b_2)$ be the second input, where G_2 is any permutation of G_1, vertex a_2 in G_2 corresponds to the vertex a_1 in G_1, and vertex b_2 in G_2 corresponds to the vertex b_1 in G_1. Then $ShortestPath(G_1,\ a_1,\ b_1)$ and $ShortestPath(G_2,\ a_2,\ b_2)$ must return the same length for the paths found.

In this section, let us consider a special kind of permutation, the *circular shift*. We regard different circular shifts as different MRs. In our experiments, we used 10-vertex graphs as test cases. As a result, we have got 9 MRs applicable to any test case, namely $Shift_1,\ Shift_2,\ \ldots,\ Shift_9$, where $Shift_i$ represents the following identity, for $i = 1,\ 2,\ \ldots,\ 9$:

$$ShortestPath(G,\ a,\ b).length = ShortestPath(\tau_i(G),\ \sigma_i(a),\ \sigma_i(b)).length, \qquad (1)$$

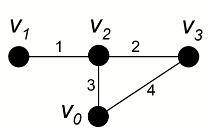
where $\tau_i(G)$ denotes the graph generated by circularly shifting left the vertices of G i times. For instance, if the vertices of G are denoted by $(v_0,\ v_1,\ \ldots,\ v_9)$, then the same vertices are denoted by $(v_2,\ v_3, \ldots,\ v_9,\ v_0,\ v_1)$ in $\tau_2(G)$. Vertex $\sigma_i(a)$ in $\tau_i(G)$ corresponds to the vertex *a* in *G*, and vertex $\sigma_i(b)$ in $\tau_i(G)$ corresponds to the vertex *b* in *G*, such as $\sigma_2(v_1) = v_3$ in our preceding example. "$ShortestPath(I).length$" denotes the path length returned by *ShortestPath* on input *I*.



(a) Graph *H*

$$\begin{pmatrix} 0 & 1 & - & - \\ 1 & 0 & 2 & 3 \\ - & 2 & 0 & 4 \\ - & 3 & 4 & 0 \end{pmatrix}$$

(b) Adjacent matrix of *H*



(c) Graph *H'*

$$\begin{pmatrix} 0 & - & 3 & 4 \\ - & 0 & 1 & - \\ 3 & 1 & 0 & 2 \\ 4 & - & 2 & 0 \end{pmatrix}$$

(d) Adjacent matrix of *H'*

Fig. 1. Graphs and their representations

The 9 MRs can be categorized into the following 3 classes:

$$\begin{aligned} Class_1 &= \{Shift_1,\ Shift_3,\ Shift_7,\ Shift_9\} \\ Class_2 &= \{Shift_2,\ Shift_4,\ Shift_6,\ Shift_8\} \\ Class_3 &= \{Shift_5\}. \end{aligned} \tag{2}$$

It is not difficult to prove that, for any 10-vertex graph G, the MRs that belong to the same class are equivalent to one another. For example, in $Class_1$, $Shift_3$ can be obtained by applying $Shift_1$ for 3 times, and $Shift_1$ can be obtained by applying $Shift_3$ for 7 times. Furthermore, any MR in $Class_1$ implies all the other MRs in $Class_2$ and $Class_3$, i.e., $R_i \Rightarrow R_j$, where $i = 1,\ 3,\ 7,\ 9$ and $j = 2,\ 4,\ 5,\ 6,\ 8$. Note that $Shift_5$ does not imply any other MR. Hence, the 4 MRs in $Class_1$ are the strongest among the 9. We shall investigate the failure-detecting capabilities of all these MRs to see whether the stronger MRs have a higher chance of revealing a failure.

The Representation of Graphs and Vertices For program $ShortestPath(G,\ a,\ b)$, the input graph G is represented by an *adjacent matrix* of size $n \times n$, where n is the number of vertices in graph G. Let us use $v_0,\ v_1,\ \ldots,\ v_{n-1}$ to denote the n vertices. If there is an edge $(v_i,\ v_j)$ in graph G, where $0 \le i,\ j < n$, then the $(i+1,\ j+1)$-entry of the adjacent matrix stores the weight of this edge; if there is no such an edge, then the $(i+1,\ j+1)$-entry of the matrix will be assigned a special value to indicate "no edge". It is also assumed that there is always an edge with weight 0 from a vertex to itself. For example, for graph H shown in subfigure (a) of Fig. 1, its adjacent matrix is shown in subfigure (b). If we want to find the shortest path between vertices v_0 and v_2 in H, then the input to program $ShortestPath(G,\ a,\ b)$ will be $G = H$, $a = 0$ and $b = 2$. Suppose this $(H,\ 0,\ 2)$ is the first test case. In metamorphic testing, if we apply an MR "circularly shift left once" to this test case, then the follow-up test case will be $(H',\ 1,\ 3)$, where H' is shown in subfigure (c) and (d) of Fig. 1. The expected relation is that the path length returned by $ShortestPath(H,\ 0,\ 2)$ and the length returned by $ShortestPath(H',\ 1,\ 3)$ must equal each other.

The Mutants To investigate the failure-detecting capabilities of the MRs, we manually seeded various faults into the source code of program *ShortestPath*. Each faulty version of the program is called a *mutant*, and each mutant has included one *simple fault*, i.e., each mutant can be turned into the correct version by making a single correction to the program. Examples of these “simple faults” are operator / operand replacements, deletion of a statement, etc. We have excluded mutants whose failures can easily be detected, such as an execution that never terminates, returning a negative path length, returning a path length of 0 when the two terminal vertices are different, and returning a nonzero path length when the two terminal vertices are identical. Furthermore, we excluded equivalent mutants using the following heuristic approach: if the outputs of two mutants are identical on all the 1000 initial test cases (which will be explained shortly), then remove one of the two mutants. In the end, we have obtained 19 mutants.

The Test Cases We first generated a set of initial test cases $T = \{t_1,\ t_2,\ \ldots,\ t_{1000}\}$. To generate this test set, we first randomly generated 50 graphs as follows: Each graph has 10 vertices $v_0,\ v_1,\ \ldots,\ v_9$. In each graph, each pair of the vertices have a 50% chance of being connected, i.e., the existence of any edge is decided by tossing a fair coin. If two vertices are connected, then the weight of the edge is randomly chosen from integers $1,\ 2,\ \ldots,\ 50$; otherwise a special value will be assigned to the corresponding entry of the adjacent matrix to indicate that the edge does not exist. For each graph thus generated, we randomly selected 20 different pairs of different nodes as the terminal vertices (note that if $(a,\ b)$ are selected, then $(b,\ a)$ will not be selected). Hence, each graph further generated 20 test cases. As a result, we have obtained a set of $20 \times 50 = 1000$ test cases $T = \{t_1,\ t_2,\ \ldots,\ t_{1000}\}$.

For each metamorphic relation $Shift_i$, where $i = 1,\ 2,\ \ldots,\ 9$, a follow-up test set $T_i = \{t_{i,1},\ t_{i,2},\ \ldots,\ t_{i,1000}\}$ was generated based on the initial test set $T = \{t_1,\ t_2,\ \ldots,\ t_{1000}\}$, where $t_{i,k}$ in T_i was a follow-up input of t_k in T, for $k = 1,\ 2,\ \ldots,\ 1000$. For each mutant program $mutant_j$, where $j = 1,\ 2,\ \ldots,\ 19$, and for each MR $Shift_i$, where $i = 1,\ 2,\ \ldots,\ 9$, $mutant_j$ was run on T and T_i, respectively. The relation of the outputs $(mutant_j(t_k),\ mutant_j(t_{i,k}))$ was checked against the MR $Shift_i$, for $k = 1,\ 2,\ \ldots,\ 1000$. Among the 1000 pairs of the outputs, if, let’s say 530 pairs did not satisfy the MR $Shift_i$, then we say the *failure rate* of $mutant_j$ against $Shift_i$ was 53%. The above procedure is described by the following pseudocode:

```
for $i$ = 1 to 9 do
  for $j$ = 1 to 19 do {
    $failureCount$ = 0;
    for $k$ = 1 to 1000 do {
      if $(mutant_j(t_k).length \neq mutant_j(t_{i,k}).length)$, where $t_k \in T$ and $t_{i,k} \in T_i$
        then $failureCount$ = $failureCount$ + 1;
      }
    Print: The failure rate of $mutant_j$ against $Shift_i$ is $failureCount / 1000$.
    }
```

The Experiment Result Our experiment result shows that, among the 9 identified MRs $Shift_1,\ Shift_2,\ \ldots,\ Shift_9$, the theoretically weakest property $Shift_5$ exhibited the highest failure-detecting capability. For clarity and ease of understanding, we grouped the 9 MRs into 3 classes according to Equation (2). Their average failure-detecting capabilities demonstrated in the experiment are shown in Fig. 2. The x-axis in the

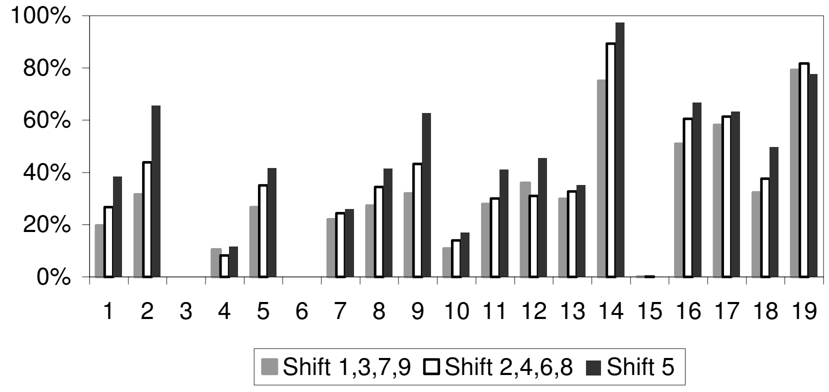


Fig. 2. A comparison of the failure-detecting capabilities of the 3 classes of metamorphic relations on the 19 mutants for the shortest path problem

figure has been divided into 19 regions. Region i of the x-axis corresponds to $mutant_i$, for $i = 1,\ 2,\ \ldots,\ 19$. The y-axis denotes the failure rate, from 0% to 100%. The histogram in region 1, for example, shows that when $mutant_1$ was tested using 1000 pairs of test cases against the MRs in $Class_1 = \{Shift_1,\ Shift_3,\ Shift_7,\ Shift_9\}$, the average failure rate was 20%; when $mutant_1$ was tested against the MRs in $Class_2 = \{Shift_2,\ Shift_4,\ Shift_6,\ Shift_8\}$, the average failure rate was 27%; when $mutant_1$ was tested against $Shift_5$, the failure rate was 39%. Note that $mutant_3$, $mutant_6$, and $mutant_{15}$ could not be killed by any MR on the test cases (the performance will be improved using our proposed MR selection strategy as will be described in Section 3). Hence, let us consider the remaining 16 mutants. Although $Shift_5$ is the weakest mathematical property among the 9 from theory because it cannot imply any other MR, it has demonstrated the highest failure-detecting capability on 15 of the 16 mutants; on the other hand, the average performance of $Class_1$ (the group of the strongest mathematical properties from theory because any MR in this group can imply all the other 8 MRs) was the worst on 13 of the 16 mutants and medium on the other 3 mutants.

This experiment has shown that theoretically stronger MRs are not necessarily good at detecting program defects. It is suggested, therefore, that selecting MRs from a pure black-box perspective is not adequate. This point is confirmed by our next case study.

2.2 A Case Study on the Critical Path Program

In project planning and scheduling, we often need to find the *critical path*, i.e., the activity that takes the longest time to complete, so that we can know what the bottleneck of the project is. Hence, the critical path problem is essentially to find the longest path in a directed and weighted graph. Let us use $CriticalPath$ to denote the program. Although it is also a graph theory program, its algorithm and data structure are totally different

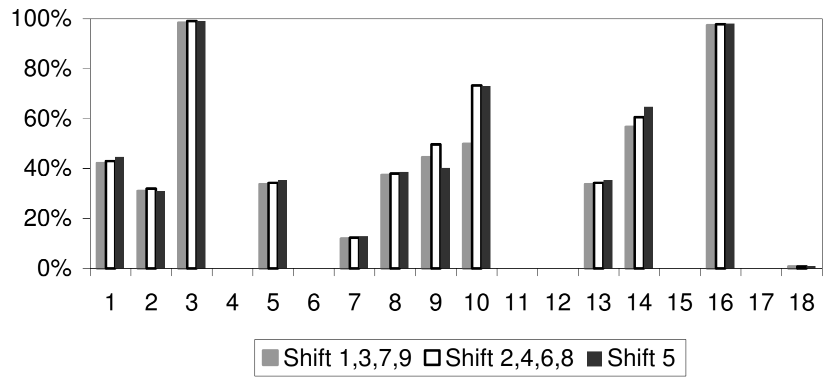


Fig. 3. A comparison of the failure-detecting capabilities of the 3 classes of metamorphic relations on the 18 mutants for the critical path problem

from the shortest path program. Hence, this program has been selected for our second case study.

The input to program *CriticalPath* is a directed and weighted graph G. If G is acyclic, the longest path in G and its length will be returned; otherwise the program will report that a cycle exists in the graph. The graph is represented by a dynamic data structure *adjacent list* (rather than the static adjacent matrix). Program *CriticalPath* is difficult to test because a practically feasible oracle is not available when the input graph is nontrivial. Hence, we have applied MT to test it.

The identification of MRs and construction of mutants and test cases were very similar to our previous case study for the shortest path problem. We used randomly generated 10-vertex directed acyclic graphs as test cases and used the 9 identified relations $Shift_1,\ Shift_2,\ \ldots,\ Shift_9$ as MRs, where $Shift_i$ has the same meaning as described previously. The graph is acyclic and represented by an array of dynamically linked lists. Again, these 9 MRs are grouped into 3 classes as in Equation (2). The initial test set T included 1000 random test cases. In addition, 18 nonequivalent mutants were created.

The experiment result is shown in Fig. 3. Among the 18 mutants, 6 could not be killed by any MR on the test cases (the performance will be improved using our proposed MR selection strategy as will be described in Section 3). Hence, let us consider the remaining 12 mutants. From the figure, we see that the performance of the 3 groups of MRs was quite close. Still, $Shift_5$ demonstrated relatively higher failure-detecting capability than the other two groups. Its failure rate ranked first on 10 of the 12 mutants, second on 1 mutant, and third on 1 mutant. On the other hand, the average performance of $Class_1 = \{Shift_1,\ Shift_3,\ Shift_7,\ Shift_9\}$ was still the worst.

As a result, our second case study confirms that the theoretically stronger MRs are not necessarily better at detecting program defects. In conclusion, selecting MRs from a pure black-box perspective is not adequate. In the next section, we shall look inside the program structure to find the reasons behind the different performance of different MRs.

3 Identifying Good MRs from a White-Box Perspective

As has been shown, theoretical knowledge of the problem domain is not adequate for distinguishing good MRs. Hence, we suggest looking into the program structure.

3.1 The Proposed MR Selection Strategy

Let p be the program under test, t be the initial successful test case, R be an MR, and t' be the follow-up test case generated according to R. For ease of presentation and understanding, let us concentrate on MRs that are identity relations. For non-identity relations, the discussion will be similar. Hence, it is the relation "$p(t) = p(t')$" that is checked in MT. Our aim is to select such an MR that has a higher chance to cause $p(t) \neq p(t')$. We propose the following hypothesis:

Hypothesis 1
For a faulty program p and a pair of metamorphic test cases $(t,\ t')$, in most situations the more the execution of $p(t')$ differs from the execution of $p(t)$, the more likely it is that their outputs are not equal.

We have not explicitly defined the concept of "difference between two executions". This concept covers all aspects of program executions, including the paths traversed, sequence of the statements exercised, sequence of different values assigned to variables, etc. Based on Hypothesis 1, our MR selection strategy is to select such MRs that can make the two executions as different as possible.

For program $p(t)$, the input t is a tuple including one or more parameters, i.e., $t = (x_1,\ x_2,\ \dots,\ x_n)$, where $n \geq 1$. Usually, different x_i's $(1 \leq i \leq n)$ play different roles in the execution and, hence, they have different influence on the overall execution flow (i.e., paths executed, variable values, iteration times, etc.) Hence, we propose selecting those MRs that can change the values of the *critical parameters* as greatly as possible. A "critical parameter" is such an x_i in t that plays the most important role in controlling how the program is to be executed. The follow-up test case t' thus generated will, therefore, force a very different execution. As a result, according to Hypothesis 1, it will be more likely that the output $p(t')$ differs from the output of $p(t)$.

Hence, our strategy considers the algorithm adopted by the programmer to be the most important factor for selecting effective MRs. Even for the same problem, an MR may have very different performance with regard to different algorithms. In the following subsections, we shall conduct case studies to test our hypothesis and the proposed MR selection strategy.

3.2 Further Study on the Shortest Path Program

Identifying More MRs The general structure of the algorithm for program $ShortestPath(G,\ a,\ b)$ is as follows: The control starts from the source vertex a. The search is conducted along the edges connected to a and will go through the vertices directly or indirectly connected to a until the destination b is reached.

According to Hypothesis 1, we identified an MR, namely *Reverse*, that was expected to be good; we also deliberately identified two more MRs, namely $Exchange(0,9)$ and *ChangeSource*, that were expected to be less effective. *Reverse* represents the property

$$ShortestPath(G,\ a,\ b).length = ShortestPath(G,\ b,\ a).length.$$

$Exchange(0,9)$ represents the property

$$ShortestPath(G,\ a,\ b).length = ShortestPath(\pi(G),\ a',\ b').length,$$

where $\pi(G)$ is a transposition of G obtained by exchanging the vertices v_0 and v_9, and a' and b' in $\pi(G)$ correspond to the vertices a and b in G, respectively. The third MR *ChangeSource* represents the property

$$ShortestPath(G,\ a,\ b).length = ShortestPath(G,\ v_i,\ b).length + d,$$

where $(a,\ v_i)$ is the first edge in the shortest path returned by $ShortestPath(G,\ a,\ b)$ and d is the weight of the edge $(a,\ v_i)$.

The MR *Reverse* was selected because we found that the *search direction* plays a critical role in the algorithm: for an input $(G,\ a,\ b)$, the algorithm always starts from a, searching along the adjacent vertices, and finish at the destination vertex b. Hence, when the source and destination are exchanged, the search sequence will be totally reversed: the control will start from b and search backwards to a. Hence, the sequence of the edges and vertices traversed in the execution of $(G,\ b,\ a)$ will be very different from that in $(G,\ a,\ b)$. According to Hypothesis 1, this MR is expected to be effective for revealing failures.

On the other hand, changing the notations of two vertices via $Exchange(0,9)$ or moving forward the starting vertex via *ChangeSource* would not have as much impact because they do not make much change to the overall execution that follows: the edges and vertices will be traversed in a similar sequence as the original execution. According to Hypothesis 1, these two MRs are expected to be less effective than *Reverse*.

Experiment Result The experiment result is shown in Fig. 4, where the mutants and the 1000 initial test cases were the same as before. Among the 19 mutants, $mutant_3$ could not be killed by any MR on the test cases. On the remaining 18 mutants, the performance of *Reverse* was obviously better than the other two: its failure rate ranked first for 15 times and second for 3 times.

We have also compared *Reverse* with $Shift_5$, the most effective MR in Section 2.1. Their average failure rate was similar, but *Reverse* can kill two mutants ($mutant_6$ and $mutant_{15}$) that could not be killed by $Shift_5$. Hence, we conclude that *Reverse* is the best MR among all the MRs studied. This experiment result supports Hypothesis 1 and shows that our MR selection strategy is effective.

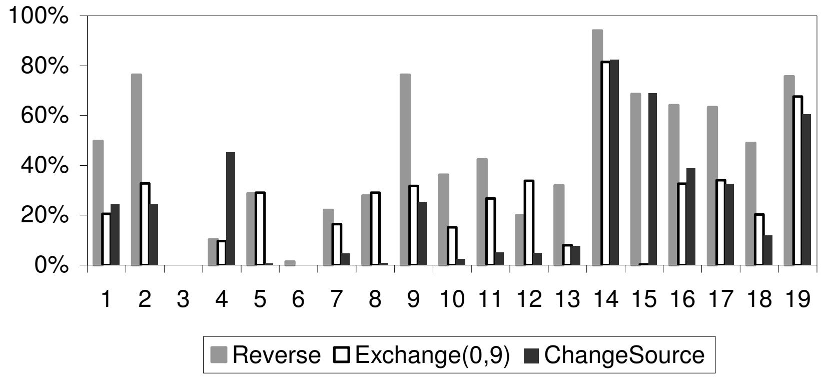


Fig. 4. Further experiment result on the shortest path program

3.3 Further Study on the Critical Path Program

Although the data structure and algorithm of program *CriticalPath* are very different from those of *ShortestPath*, we found that the "search direction" is still the critical factor. This is understandable because almost all searching algorithms are performed along a certain direction. In a graph, the search usually starts from the source vertex and go along the adjacent edges towards the destination vertex. If the search direction is changed greatly, the execution sequence will be changed greatly as well and, as a result, the second output will be more likely to differ from the first one. For program *CriticalPath*, the source vertex is the one whose in-degree is 0; the destination vertex is the one whose out-degree is 0. According to Hypothesis 1, we have identified the MR *ChangeDirection* : $CriticalPath(G).length = CriticalPath(G').length$, where G' is obtained by reversing the directions of all the edges in G.

We would like to compare the failure-detecting capabilities of the 3 MRs *ChangeDirection*, $Exchange(0,9)$, and $Shift_5$, as shown in Fig. 5. Among the same 18 mutants and on the same 1000 initial test cases, $mutant_4$ and $mutant_6$ could not be killed by any MR on the test cases. For the remaining 16 mutants, *ChangeDirection* has killed all of them, but $Shift_5$ and $Exchange(0,9)$ could only kill 12 of them. For the 12 mutants that can be killed by all these MRs, the failure rate of *ChangeDirection* ranked first for 10 times and slightly lower than $Shift_5$ on $mutant_3$ and $mutant_{16}$.

3.4 Why Does It Work?

We have also studied individual cases for the reasons why our proposed strategy works. Because of the length limit of this paper, we shall only describe the rationale behind. For program $p(x)$ implementing function $f(x)$, let t be the first test case and t' be the follow-up test case generated with regard to an MR. Suppose $p(t)$ has not revealed any failure, then there are actually two possibilities: (1) $p(t) = f(t)$; (2) $p(t) \neq f(t)$ but

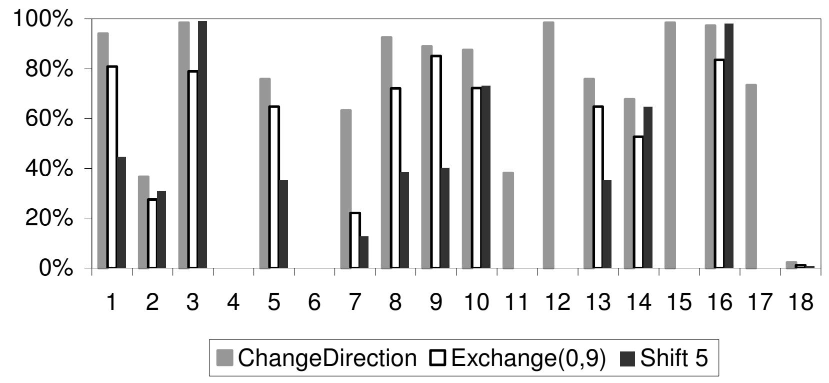


Fig. 5. Further experiment result on the critical path program

this could not be detected by the tester because of the lack of the oracle. For case (1), there are two subcases: (1.1) $p(t)$ did not touch the buggy code; (1.2) $p(t)$ touched the buggy code but the output happened to be correct. For case (1.1), obviously the more the execution of $p(t')$ differs from that of $p(t)$, the higher is the chance for $p(t')$ to touch the buggy code and, hence, to reveal a failure; For case (1.2), if the execution of $p(t')$ is very similar to that of $p(t)$, then the cause that made $p(t) = f(t)$ may still remain in the execution of $p(t')$ and, as a result, $p(t')$ may also compute correctly; For case (2), the reasoning is similar: the greater the similarity between the execution of $p(t)$ and $p(t')$, the higher the chance for both executions to make the same error and, hence, output the same result. In this situation, although both outputs are wrong with regard to function f, the failure cannot be detected with regard to the MR.

4 Discussions and Conclusion

Metamorphic testing method effectively exploits successful test cases and alleviates the oracle problem. Since the procedure is straightforward and can be fully automated, MT is cost-efficient and, hence, useful and practical for practitioners.

For many problems, more than one MR can be identified. It would be ideal if all MRs could be used for testing. Since the resources for software development are always limited, however, it is desirable to know which MRs should be given priority for use in testing. In this article, we conducted case studies using the mutants of two programs in graph theory, where there is an oracle problem. The main results are: (a) theoretical knowledge of the problem domain is not adequate for distinguishing good MRs, and (b) good MRs should be those that can make the multiple executions of the program as different as possible.

Strictly speaking, our MR selection strategy emphasizes the importance of the structure of the program under test. However, it is not practical to require the testers to fully understand the program code before testing. Hence, we propose that good MRs should be selected with regard to the *algorithm* that the program follows because algorithms are easier to understand than the source code. Although the programmer may make mistakes in the implementation, the general structure of the algorithm should be kept because of the competent programmer hypothesis.

It must be pointed out that: (1) MT is a technique for generating *follow-up* test cases. In other words, pure MT is not adequate for software quality assurance. It must be combined with other test case selection strategies. (2) Our experiment result shows that different MRs have different failure-detecting capabilities with regard to different types of program defect. How to employ different MRs in a collaborative and complementary way to achieve the best result will be a future research topic.

We have not defined the concept of "difference between two executions" explicitly because the execution of programs is very complicated. We shall study this issue and give more explicit guidelines in our future research. In addition, we shall look seriously into the phenomenon that some mutants could not be killed by any identified MR on the test cases.

References

1. Hailpern, B., Santhanam, P.: Software debugging, testing, and verification. IBM Systems Journal **41** (2002) 4–12
2. Beizer, B.: Software Testing Techniques. Van Nostrand Reinhold, New York (1990)
3. Howden, W. E.: Reliability of the path analysis testing strategy. IEEE Transactions on Software Engineering (1976) 208–215
4. Weyuker, E. J.: On testing non-testable programs. The Computer Journal **25** (1982) 465–470
5. Hamlet, D.: Predicting dependability by testing. In: Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 1996), ACM Press, New York (1996) 84–91
6. Manolache, L. I., Kourie, D. G.: Software testing using model programs. Software: Practice and Experience **31** (2001) 1211–1236
7. DeMillo, R. A., Lipton, R. J., Sayward, F. G.: Hints on test data selection: help for the practicing programmer. IEEE Computer **11** (1978) 34–41
8. Myers, G. J.: The Art of Software Testing. Wiley, New York (1979)
9. Morell, L. J.: A theory of fault-based testing. IEEE Transactions on Software Engineering **16** (1990) 844–857
10. Chen, T. Y., Cheung, S. C., Yiu, S. M.: Metamorphic testing: a new approach for generating next test cases. Technical Report HKUST-CS98-01, Department of Computer Science, Hong Kong University of Science and Technology, Hong Kong (1998)
11. Chen, T. Y., Feng, J., Tse, T. H.: Metamorphic testing of programs on partial differential equations: a case study. In: Proceedings of the 26th Annual International Computer Software and Applications Conference (COMPSAC 2002), IEEE Computer Society Press, Los Alamitos, California (2002) 327–333
12. Chen, T. Y., Tse, T. H., Zhou, Z. Q.: Fault-based testing without the need of oracles. Information and Software Technology **45** (2003) 1–9
13. Gotlieb, A., Botella, B.: Automated metamorphic testing. In: Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC 2003), IEEE Computer Society Press, Los Alamitos, California (2003) 34–40

14. Cody, Jr, W. J., Waite, W.: Software Manual for the Elementary Functions. Prentice Hall, Englewood Cliffs, New Jersey (1980)
15. Ammann, P. E., Knight, J. C.: Data diversity: an approach to software fault tolerance. IEEE Transactions on Computers **37** (1988) 418–425
16. Blum, M., Kannan, S.: Designing programs that check their work. In: Proceedings of the 31st Annual ACM Symposium on Theory of Computing (STOC'89), ACM Press, New York (1989) 86–97
17. Blum, M., Kannan, S.: Designing programs that check their work. Journal of the ACM **42** (1995) 269–291
18. Blum, M., Luby, M., Rubinfeld, R.: Self-testing / correcting with applications to numerical problems. Journal of Computer and System Sciences **47** (1993) 549–595