2013

On the Correlation between the Effectiveness of Metamorphic Relations and Dissimilarities of Test Case Executions

Yuxiang Cao
University of Wollongong, yc962@uowmail.edu.au

Zhi Quan Zhou
University of Wollongong, zhiquan@uow.edu.au

Tsong Yueh Chen
Swinburne University of Technology, tchen@ict.swin.edu.au

Publication Details

Cao, Y., Zhou, Z. & Chen, T. (2013). On the Correlation between the Effectiveness of Metamorphic Relations and Dissimilarities of Test Case Executions. QSIC 2013: 13th International Conference on Quality Software (pp. 153-162). Nanjing, China: IEEE.

On the Correlation between the Effectiveness of Metamorphic Relations and Dissimilarities of Test Case Executions

Abstract
Metamorphic testing (MT) is a property-based automated software testing method. It alleviates the oracle problem by testing programs against metamorphic relations (MRs), which are necessary properties among multiple executions of the target program. For a given problem, usually more than one MR can be identified. It is therefore of practical importance for testers to know the nature of good MRs, that is, which MRs are likely to have higher chances of revealing failures. To address this issue we investigate the correlation between the faultdetection effectiveness of MRs and the dissimilarity (distance) of test case execution profiles. Empirical study results reveal that there is a strong and statistically significant positive correlation between the fault-detection effectiveness and the distance. The findings of this research can help to develop automated means of selecting/prioritizing MRs for cost-effective metamorphic testing.

Keywords
executions, metamorphic, case, effectiveness, between, correlation, test, dissimilarities, relations

Disciplines
Engineering | Science and Technology Studies

Publication Details
Cao, Y., Zhou, Z. & Chen, T. (2013). On the Correlation between the Effectiveness of Metamorphic Relations and Dissimilarities of Test Case Executions. QSIC 2013: 13th International Conference on Quality Software (pp. 153-162). Nanjing, China: IEEE.

On the Correlation between the Effectiveness of Metamorphic Relations and Dissimilarities of Test Case Executions

Yuxiang Cao
School of Computer Science and Software Engineering
University of Wollongong
Wollongong, NSW 2522, Australia

Zhi Quan Zhou *
School of Computer Science and Software Engineering
University of Wollongong
Wollongong, NSW 2522, Australia

Tsong Yueh Chen
Faculty of Information & Communication Technologies
Swinburne University of Technology
Hawthorn, VIC 3122, Australia

***Abstract*—Metamorphic testing (MT) is a property-based automated software testing method. It alleviates the oracle problem by testing programs against metamorphic relations (MRs), which are necessary properties among multiple executions of the target program. For a given problem, usually more than one MR can be identified. It is therefore of practical importance for testers to know the nature of good MRs, that is, which MRs are likely to have higher chances of revealing failures. To address this issue we investigate the correlation between the fault-detection effectiveness of MRs and the dissimilarity (distance) of test case execution profiles. Empirical study results reveal that there is a strong and statistically significant positive correlation between the fault-detection effectiveness and the distance. The findings of this research can help to develop automated means of selecting/prioritizing MRs for cost-effective metamorphic testing.**

Keywords: Software testing, metamorphic testing, metamorphic relation, fault-detection effectiveness, execution dissimilarity, distance measurement, initial execution, follow-up execution.

I. Introduction

In the literature of software testing, it is generally assumed that a test oracle exists, which is known as the *oracle assumption*. This assumption, however, does not always hold. In some situations, it is very expensive or even impossible to decide whether an output of a test case execution is correct, which is known as the *oracle problem*; in other situations, even if an oracle is available, if it cannot be automated, the manual predictions and verifications of outputs can often be time consuming and error prone [1].

A *metamorphic testing* (MT) method has been proposed to alleviate the oracle problem [2], [3]. MT is both a technique for automated test case generation and a mechanism for automated result verification, through the use of some expected properties of the target program. These properties are known as *metamorphic relations* (MRs), which are necessary relations among the inputs and outcomes of multiple executions of the target program. For the sine function, for instance, many MRs can be identified using the domain knowledge about trigonometric functions, such as

i) sin(x) = sin(x+360)
ii) sin(x) = -sin(x+180)
iii) sin(x) = -sin(-x)
iv) sin(x) = sin(180-x)
...

To test a program $p(x)$ that implements sine function, a test suite $T = t_1, t_2, \ldots, t_n$, where $n \geq 1$, is generated using certain strategies such as branch coverage testing, category-partition testing, or just random testing. If no failure is detected after running every element of T, T is said to be a set of *successful test cases*, which is normally retained for future regression testing only. MT, however, proposes that, T (which is called a set of *initial test cases*) can in fact be exploited to generate a set of *follow-up test cases* and hence the program can be automatically further tested. To do so, MT needs to refer to a metamorphic relation (MR). Without loss of generality, let us say the selected MR is "sin(x) = -sin(x + 180)". Then a follow-up test suite $T' = \{t'_1, t'_2, \ldots, t'_n\}$ can be generated, where $t'_i = t_i + 180$, $i = 1, 2, \ldots, n$. The program can then be tested on T' and the outputs can be verified automatically against the MR. If, after taking into consideration the acceptable rounding errors in floating-point arithmetic, $p(t) \neq p(t'_i)$ for some i, then a failure is immediately revealed.

Note 1: A metamorphic test involves the executions of the *initial* and the *follow-up* test cases, hence running the program under test more than once.

Note 2: An MR is a necessary property identified from the problem domain, and is usually not sufficient for program correctness. This is, however, the limitation of all testing methods.

* All correspondence should be addressed to Dr. Zhi Quan Zhou, School of Computer Science and Software Engineering, University of Wollongong, Wollongong, NSW 2522, Australia. Email: zhiquan@uow.edu.au. Telephone: (61-2) 4221 5399.

Note 3: There are related techniques known as program checker [4] and self-testing/correcting [5], [6], which make intensive use of expected identity relations of the target functions to test programs and check outputs automatically and probabilistically. MRs, however, are not limited to identity relations. For example, Zhou et al. [7] identified a set of non-identity MRs to alleviate the oracle problem in testing Web search engines. MRs are also used for debugging [8], [9] and for fault-based testing [10].

Note 4: In the sine example, the value of the follow-up test case *only* depends on the value of the initial test case. In other situations, the input values may also depend on the *output* values. For example, consider a program $q(x,\ y)$ that calculates $x \times y$. To test this program, the associative law $(a \times b) \times c = a \times (b \times c)$ can be identified as an MR. Using this MR, a metamorphic test will execute program q 4 times; the test cases (input values) of these 4 executions not only relate to each other but also relate to the output values of the other executions.

For a given problem, normally more than one MR can be identified. It would be ideal if all MRs can be used for testing. However, since resources are always limited, testers need practical guidance to know which MRs should be given priority for use in testing. Therefore, selection of effective MRs that have higher chances of detecting failures is a key focus of MT research. A pilot study on the effectiveness of MRs [11] shows that MRs whose *initial execution* (that is, execution on an initial test case) and *follow-up execution* (that is, execution on a follow-up test case) are very different are likely to have a higher chance of detecting failures than those whose initial and follow-up executions are similar. The concept of "difference" or "dissimilarity" between executions, however, is not clearly defined in Chen et al. [11]; this could include, for example, execution paths, data flows, coverages, etc.

Recently, Zhou et al. [12], [13] proposed to measure the distance (or difference) between test cases using the concepts of *coverage Manhattan distance* (CMD), *frequency Manhattan distance* (FMD), and *frequency Hamming distance* (FHD) in order to conduct *adaptive random testing* (ART). Among the investigated metrics, they found that the CMD metric based on branch coverage execution profiles had the best fault-detection effectiveness. Zhou et al.'s work [12], [13] was limited to ART and did not include any study on metamorphic testing.

The research questions of this paper is stated below: *Can the distance metrics based on execution profiles ([12], [13]) also be used to quantitatively measure the dissimilarity of test case executions in metamorphic testing? If yes, is there a strong correlation between the distance measures and the fault-detection effectiveness of metamorphic relations?*

A positive answer to the above research question will help to reveal the nature of "good" MRs, which will provide a hint for practicing software testers to better select and prioritize MRs. Answers to the research question may also help to develop automated means for the selection and prioritization of MRs.

The rest of this paper is organized as follows: Section II introduces the ideas and some basic concepts of this paper. Section III describes the designs of experiments. Section IV reports the experimental results, and Section V further compares with related work. Section VI discusses future research topics such as applications of the findings of this research to software testing in practice and concludes the paper.

II. Basic Ideas and Concepts

A. The dissimilarity between initial and follow-up test case executions in metamorphic testing

Consider the selection of effective MRs in metamorphic testing from a white-box perspective. First, we wish to discuss why some MRs are not effective, that is, why failures cannot be detected using some MRs. Let p be a faulty program under test, which purportedly implements function f. Let t be a test case and suppose that $p(t)$ did not reveal a failure. Note that it may not necessarily mean that $p(t) = f(t)$ because it is possible that $p(t) \neq f(t)$ but the tester is unable to identify this failure owing to the lack of a test oracle. Without the need of an ideal oracle, let us test p against a metamorphic relation R_1. There are many kinds of MRs, and in this paper we focus on identity relations for ease of presentation. For example, let R_1 have the following simplest form: $f(x) = f(x')$ (such as sin(x) = sin(180-x)). For more comprehensive MRs and those involving more than two executions, the discussion is similar. Let t' be the follow-up test case corresponding to the initial test case t, according to R_1. Suppose that this metamorphic test (which involves an "initial execution" $p(t)$ and a "follow-up execution" $p(t')$) did not reveal a failure, that is, $p(t) = p(t')$. Then there are 3 possible causes for this phenomenon: (1) Both the initial execution and the follow-up execution did not touch the defective part of the code and, therefore, both outputs are correct. (2) Only one of the two executions, say $p(t)$, touched the defective part of the code but the output happened to be correct, that is, $p(t) = f(t)$ coincidentally. (3) Both executions touched the defective code, but $p(t)$ still equals $p(t')$, that is, although both outputs may be wrong, the identity relation still holds.

Case (1) suggests that if the faulty program p executes t and t' in a similar fashion (for example, both t and t' execute the same path), then the chance for the two outputs to agree with each other (hence, no failure can be detected) will be higher than that where t and t' are executed differently. This analysis also applies to case (3): if the two executions have been done in a similar fashion then, intuitively, the chance of producing consistent outputs will be higher than that for very different executions. With regard to case (2), it is by chance and will not be discussed in this paper.

Note that MRs are used to generate follow-up test cases. Therefore, if an MR, say R_2, can make the *follow-up execution* more different from the *initial execution* than could R_1, then, intuitively, R_2 could have a better fault-detection capability than R_1. Now the key issue is: how to measure the "difference" between the initial and the follow-up executions? People (such

as the designer, programmer or tester) who are familiar with the specification or algorithm of the program under test should have some idea about this difference because they normally have some general idea about how the program will be run (e.g. how the program's control flow will be exercised) on different kinds of inputs. In this paper, however, we are more interested in quantitative rather than qualitative approaches.

B. Execution profiles: abstraction of test case executions

An execution profile records some aspects of a program's execution. For example, a branch profile records the execution coverage or frequency of each branch in a run. In the software engineering literature, the concept of execution profiles has been widely used, such as in observation-based testing [14], [15] and regression testing [16], [17]. Many aspects of program execution can be profiled, such as the control flow, data flow, variable values and event sequences. In this research, we focus on the *statement profile* and the *branch profile*. A statement profile records the number of times that each statement is executed during an execution run, which consists of a vector of counts, with one count per statement in the program. A branch profile records the number of times that each branch is executed during an execution run, which also consists of a vector of counts, with one count per branch in the program. In a program, at each decision point, there are two branches, namely, a true and a false branch.

C. Distance metrics: a proposal to measure the dissimilarity between initial and follow-up executions in MT

We propose to quantitatively measure the dissimilarity between initial and follow-up executions of MT by calculating the distance between their execution profiles. We hypothesize that the larger the distance between the initial and the follow-up execution profiles, the more capable the MR will be in detecting failures. If this hypothesis is found to be valid, then we will be able to provide practitioners with a quantitative approach to measuring MRs for metamorphic testing and even for reliability estimate (which will be briefly discussed in Section VI).

We will study the correlation between distance metrics and the fault-detection effectiveness of MRs. The following 3 distance metrics are adopted to measure the distance (or difference) between the initial and follow-up executions in MT: the *coverage Manhattan distance* (CMD) [12], the *frequency Manhattan distance* (FMD) [13] and the *frequency Hamming distance* (FHD) [13].

The CMD metric only concerns whether a statement or a branch has been covered without counting the frequency. If a statement or branch has been executed at least once, the flag for that statement or branch in the execution profile is set to 1; otherwise it is set to 0. Let x and x' be an initial and a follow-up test case, respectively. Let $X = (x_1, x_2, \ldots, x_n)$ and $X' = (x'_1, x'_2, \ldots, x'_n)$ be the execution profiles of x and x', respectively. The CMD between X and X' is calculated as: $\mathit{CMD}(X,\ X') = \sum_{i=1}^{n} |x_i - x'_i|$, where n is the number of statements or branches, and the values of x_i and x'_i ($i = 1, 2, \ldots, n$) are either 1 or 0. When branches are considered, it is written as BCMD; when statements are considered, it is written as SCMD.

The FMD metric concerns frequency. Let $X = (x_1, x_2, \ldots, x_n)$ and $X' = (x'_1, x'_2, \ldots, x'_n)$ be the execution profiles of the initial test case x and the follow-up test case x', respectively, where x_i and x'_i ($i = 1, 2, \ldots, n$) are the number of times (that is, frequency) that statement or branch i has been exercised by the corresponding test case. FMD compares each $(x_i,\ x'_i)$ pair ($i = 1, 2, \ldots, n$) and sums up the differences: $\mathit{FMD}(X,\ X') = \sum_{i=1}^{n} |x_i - x'_i|$. When branches are considered, it is written as BFMD; when statements are considered, it is written as SFMD.

The FHD metric is also based on frequency. It concerns how many $(x_i,\ x'_i)$ pairs are not identical regardless of how large the difference is: $\mathit{FHD}(X,\ X') = \sum_{i=1}^{n} k_i$, where $k_i = 0,$ if $x_i = x'_i$, otherwise $k_i = 1$, and x_i and x'_i are the number of times that statement or branch i has been exercised by the corresponding test case, $i = 1, 2, \ldots, n$. When branches are considered, it is written as BFHD; when statements are considered, it is written as SFHD.

III. Design of Experiments

A series of experiments with 7 subject packages have been conducted to answer the research question stated in Section I. This section descries the experimental design including dependent and independent variables, subject programs, the coverage monitoring tool, and the experimental procedure.

A. A summary of dependent and independent variables in the experiments

Independent variables of the experiments include:
1) Distance metric, which has the following values: SCMD, SFHD, SFMD, BCMD, BFHD, and BFMD (that is, CMD, FHD and FMD for statements and branches).

2) Program under test, test case, and MR. A total of 7 subject packages are used, namely, *spWiki*, *cpWiki*, *spStudent*, *bigInt*, *grep*, *sed*, and *bash*. Each package contains a set of faulty programs (which include real or seeded faults), a test suite and a set of MRs. The *spStudent* and *bigInt* packages contain small programs with real faults; *spWiki* and *cpWiki* are packages containing small programs with seeded faults; *grep* and *sed* are packages containing medium to large programs with seeded or real faults. *bash* is a package containing large programs with seeded faults.

Dependent variables of the experiments include:
1) Distance, which has a non-negative value. For each faulty program under test and each of the 6 distance metrics and each MR, a distance value will be calculated after each metamorphic test (which involves initial and follow-up executions) and, after all the metamorphic tests against the given MR have been completed, a mean distance will be

calculated with respect to the given faulty program, the given distance metric and the given MR.

2) Failure-detection rate, which has a real value in the range of $[0,\ 1]$. For each faulty program and each MR, after all the metamorphic tests have been completed, a failure-detection rate r is calculated to indicate the fault-detection effectiveness of the given MR for the given faulty program. For example, if, out of a total of 100 metamorphic tests (where each metamorphic test involves one initial and one or more follow-up test case executions), 2 violations of the MR are detected, then the failure-detection rate (which means the violation rate) is 0.02. Note that different MRs may have different failure-detection rates. $r = 0$ means that no violation of the MR can be detected; $r = 1$ means that each and every metamorphic test can detect a violation.

3) Correlation coefficient between mean distance and failure-detection rate, which has a real value in the range of $[-1, 1]$. For each faulty program and each of the 6 distance metrics, a correlation coefficient is calculated to measure the correlation between the fault-detection effectiveness (measured by the failure-detection rates of different MRs) and the dissimilarities of initial and follow-up executions (measured by the mean distances of different MRs). As a result, for each faulty program, after all the metamorphic tests have been completed, 6 correlation coefficients will be calculated, corresponding to the 6 distance metrics. An example of calculating such a correlation coefficient will be given later in the paper.

B. Subject programs, MRs, and test cases

Subject programs are summarized in Table I. There is a total of 7 packages classified into 3 groups according to their sizes. Each package contains a set of faulty programs, a set of MRs, and a set of metamorphic test cases.

1) spWiki: The first program, *spWiki*, implements Dijkstra's shortest path algorithm to find the shortest path between a source vertex a and a destination vertex b in graph G, where G is an undirected graph with positive edge weights. For nontrivial input, it is not easy to verify the output. The program was written by a master's student based on the pseudocode available in http://en.wikipedia.org/wiki/Dijkstra's_algorithm . Here Wikipedia was not used as a rigorous information source, but was used for creating programs to test. After creating the program, the student was asked to manually seed faults into the code in such a way that the faults should be as realistic as possible. Faulty versions that easily crash were excluded from the experiments. Finally, 19 faulty *spWiki* programs were collected.

The main data structure of the program is the input graph G with n vertices represented by an $n \times n$ adjacency matrix. The matrix is symmetric and has 0's on its diagonal.

We first generated an initial set of 1,000 random test cases. This was achieved by randomly generating 50 undirected 10-vertex graphs, where the maximum edge weight was 50. For each graph, 20 different pairs of source and destination vertices were randomly chosen where the source and destination vertices were not the same. For two different vertices a and b, if $(a,\ b)$ is chosen, then $(b,\ a)$ will not be selected again. In this way, we obtained $20 \times 50 = 1,000$ test cases. This is the set of *initial test cases* for all MRs.

To test the program without the need of an ideal oracle, MT can be applied. It is not difficult to identify MRs using the knowledge of the problem domain [11]. These MRs are listed below:

i) reverse: The follow-up test case is generated by reversing the source and destination vertices while the graph remains the same. The expected relation is that the lengths of the paths returned by the initial and follow-up executions should be the same. That is, *ShoretestPath*$(G,\ a,\ b)$.*length* = *ShortestPath*$(G,\ b,\ a)$.*length*.

ii) exchange: Let $\pi(G)$ be a transposition of graph G obtained by exchanging two and only two vertices in G. Hence, G and $\pi(G)$ are isomorphic but have different adjacency matrices. Let a' and b' be the vertices in $\pi(G)$ corresponding to the vertices a and b in G, respectively. The MR is: *ShortestPath*$(G,\ a,\ b)$.*length* = *ShortestPath*$(\pi(G),\ a',\ b')$.*length*. We define a to be vertex 0 of graph G. Thus, we have got 9 MRs, denoted by exchange(0, 1), exchange(0, 2), ..., exchange(0, 9).

iii) shift: This MR is similar to "exchange" in that both of them apply a kind of permutation to the vertex vector of G and check whether the returned lengths are equal. The MR "shift" circularly shifts left this vector to generate an isomorphic graph G' that has a different adjacency matrix. Since G has 10 vertices, we have got 9 different MRs, namely, $shift_1$, $shift_2$, ..., $shift_9$, which means circularly shifting left the vertices of G once, twice, ..., 9 times, respectively. Note that $shift_{10}$ is equivalent to the original vector and therefore is not included.

iv) scale: In the follow-up test case, we double the weight of each edge. The expected relation is that the length of the returned path should also be doubled.

2) cpWiki: The second program, *cpWiki*, finds the *critical path* in a directed graph G [11]. The algorithm is often used in project planning and scheduling to find the most time-consuming chain of activities. The set of activities and the scheduling constraints are represented using a weighted directed acyclic graph. The weights can be either on the vertices or on the edges. In our experiment we used edge-weighted graphs. Although such a graph must be acyclic, the programs under test can receive cyclic graphs. When the input graph G is cyclic, the program will return a special value, -1, to indicate this situation. The program development and fault injection process was similar to that of *spWiki*. Note that, although *cpWiki* is also a graph theory program, its algorithm is different from that of *spWiki*. Further, instead of using the adjacency matrix, the input graph is represented using a dynamic structure, namely, the adjacency list.

TABLE I
BASIC INFORMATION OF SUBJECT PROGRAMS

group	package	programming language	fault type	# of faulty programs	source lines of code	origin	# of MRs
small	spWiki	C	seeded	19	95	adapted from wikipedia	20
	cpWiki	C	seeded	18	125	adapted from wikipedia	20
	spStudent	C++	real	10	avg 200	university assignments	20
	bigInt	C++	real	21	avg 500	university assignments	43
medium to large	grep	C	seeded	5	10068	SIR	10
	sed	C	real, seeded	7	14427	SIR	33
large	bash	C	seeded	6	59846	SIR	10

The identified MRs and the generation of test cases are very similar to those of *spWiki*. When "reverse" is applied to *cpWiki*, it means "change direction," that is, the directions of all edges of the initial graph G are reversed. Other MRs include exchange, shift and scale, as explained in Section III-B1.

3) spStudent: The third program, *spStudent*, refers to a collection of programming assignments. The assignments were submitted by university students in their second year who were doing a programming subject for their 3-year bachelor of computer science degree. The program attempts to find both the shortest and the second shortest path between two vertices in a graph. A total of 10 C++ assignments at a pass grade or higher were collected. The average size is about 200 SLOC (that is, source lines of code excluding comments and blanks).

The identified MRs and the generation of test cases are very similar to those of *spWiki* as explained in Section III-B1. Note that there were no manually seeded faults in the programs.

4) bigInt: The fourth program is named *bigInt*, which implements multiple precision arithmetic, that is, it is a calculator for very large integers. The subject programs were collected from year 2 assignments of university students who were doing a software engineering subject. The *bigInt* program is run in the command line, where the user enters a simple mathematical expression, and the evaluated result is printed. A "simple mathematical expression" includes integers (which can be either positive or negative), $+$, $-$, $\times$ and $/$ (integer division). The length of a valid expression is between 1 and 50 characters. A total of 21 students' C++ assignments (at a pass grade or higher) were collected. The average size is about 500 SLOC. When performing MT on these 21 programs, failures (violations of MRs) were detected for each and every program. Note that there were no manually seeded faults in the programs and, hence, all faults were real. Details of test case generation and MRs are described below.

To construct a set of initial test cases that can be used for all MRs, we first generated a set S that contains 1,000 pairs of randomly generated multiple precision integers (to serve as operands for all MRs), that is, $S = \{(a_1,\ b_1), (a_2,\ b_2), \ldots, (a_{1000},\ b_{1000})\}$. A rule is that for any pair $(a_i,\ b_i)$ in S, the following pairs will *not* appear in S: $(b_i,\ a_i)$, $(-a_i,\ -b_i)$, and $(-b_i,\ -a_i)$.

It is not difficult to identify some MRs for *bigInt*, which are given below:

i) $a + b = b + a$.
ii) $a \times b = b \times a$.
iii) $(a+b)+x = (x+b)+a$, where $(a,\ b)$ are from set S and x is taken from X, and X is a set containing 20 randomly generated multiple precision integers with unique values. Because there are 20 different values for x, this MR has 20 different sub-MRs.
iv) $x \times (a+b) = (a \times x)+(b \times x)$, where $(a,\ b)$ are from set S and x is taken from X. Because there are 20 different values for x, this MR has 20 different sub-MRs.
v) $a + b = -((-a) + (-b))$.

As a result, there is a total of 43 MRs (including the sub-MRs).

5) grep: Grep is a command-line utility that searches the input file(s) for lines containing a match to the given pattern in the form of a regular expression. By default, grep prints the matching lines. Grep was originally developed for Unix but now is available for all Unix-like systems. The *grep* package we used was downloaded from the Software-artifact Infrastructure Repository (SIR, http://sir.unl.edu) [18]. The package includes both base versions and faulty versions together with test suites. The size of the program is about 10,068 SLOC. A total of 5 faulty programs were used in our experiment, corresponding to the following *grep* versions: v2.2, v2.3, v2.4, v2.4.1, and v2.4.2. We used the default faults already activated when the package was downloaded.

We constructed a set of 10,000 initial test cases as follows: We first extracted the regular expressions from 870 test cases included in the *grep* package, and then added another group of randomly generated regular expressions to get a total of 1,000 regular expressions. Then, we used the source code files of 10 large programs as input files (3 are included in the *grep* package, and 7 are downloaded from SIR including *bash*, *flex*, *sed*, *space*, *vim*, *gzip*, *make*). In this way, we obtained a total of $1,000 \times 10 = 10,000$ initial test cases. Each of these 10,000 test cases includes all the necessary components required by the identified MRs. These MRs were identified using knowledge of regular expressions, as explained below. Note that in real-world systems different grep versions may vary in the types of regular expressions that they support. All the MRs listed below (as well as those for the *sed* and *bash* programs that will be introduced later) have been validated against the base version programs in the package.

i) MR1: The range operator (such as [0-9]) is equivalent to an expression that enumerates all its elements (such as

[0123456789]).

ii) MR2: The range operator (such as [0-9]) is equivalent to an expression that enumerates all its elements separated by \| (such as [0\|1\|2\|3\|4\|5\|6\|7\|8\|9]).

iii) MR3: Some range operators are equivalent to certain predefined classes of characters. For example, [0-9] is equivalent to [[:digit:]].

iv) MR4: In this MR, the initial execution involves a range operator, such as [0-9]; and there are two follow-up executions connected by a pipe (|) operator where the first follow-up execution makes use of the [^] expression that means "to match any one character except those enclosed in [ ]." For example, the following (initial) execution

```
grep '[0-9]' myFile.txt
```

and the following (follow-up) executions

```
grep '[^[:alpha:]]' myFile.txt | grep '[[:alnum:]]'
```

should be equivalent. Another example is:

```
grep '[a-zA-Z]' myFile.txt
```

and

```
grep '[^[:digit:]]' myFile.txt | grep '[[:alnum:]]'
```

should be equivalent. Two examples (instances) of this MR are shown in Figure 1. Because each metamorphic test involves 3 executions of *grep*, there can be 3 execution profile distances, namely, $d(exe_1, exe_2)$, $d(exe_1, exe_3)$, and $d(exe_2, exe_3)$. The maximum of these 3 distances is recorded.

v) MR5: The symbols "\|" can be inserted into a range expression to create an equivalent expression. For example, [0123456789] is equivalent to [0\|1\|2\|3\|4\|5\|6\|7\|8\|9].

vi) MR6: Certain predefined classes of characters are equivalent to a range expression that enumerates all its elements. For example, [[:digit:]] and [0123456789] are equivalent.

vii) MR7: Same as MR4, except that the range operator (such as [0-9]) is changed to a range expression that enumerates all its elements (such as [0123456789]).

viii) MR8: Certain predefined classes of characters are equivalent to a range expression that enumerates all its elements involving the symbols "\|." For example, [[:digit:]] and [0\|1\|2\|3\|4\|5\|6\|7\|8\|9] are equivalent.

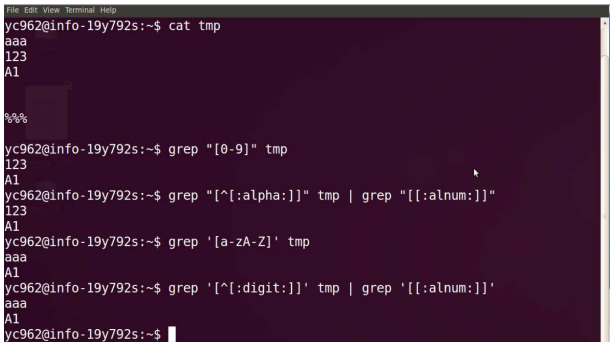


Fig. 1. Linux screenshot: two instances (examples) of *grep*'s MR4

ix) MR9: Same as MR4, except that the range operator (such as [0-9]) is changed to a range expression involving the symbols "\|" (such as [0\|1\|2\|3\|4\|5\|6\|7\|8\|9]).

x) MR10: Same as MR4, except that the range operator (such as [0-9]) is changed to certain predefined classes of characters (such as [[:digit:]]).

6) sed: Sed is another popular Unix utility often used as a subject program in software engineering empirical studies. It is a stream editor that performs text transformations on an input stream.

The *sed* package used in our experiment was also downloaded from the SIR site [18]. The package includes both base versions and faulty versions together with test suites. The size of the program is about 14,427 SLOC. A total of 7 faulty programs were used in the experiment, corresponding to the following *sed* versions: v1.18, v2.05, v3.01, v3.02, v4.0.6, v4.0.7 and v4.1.5. According to the SIR site, the faulty versions include both real and seeded faults. We used the default faults already activated when the package was downloaded.

Using a method similar to that of *grep*, we obtained a set of 4,333 initial test cases and a set of 33 MRs. The 4,333 initial test cases are valid for all the 33 MRs. All of the MRs make use of equivalence relations between different forms of regular expressions. Because these MRs are similar to those of *grep*, they are not listed in this paper to avoid undue lengthiness.

7) bash: The 7th subject program is *bash*, a large program that has about 59,846 SLOC. Bash is an sh-compatible shell, or command language interpreter, of the GNU operating system. Bash incorporates useful features from the Korn shell and C shell, and offers functional improvements over sh for both the programming interface and the interactive user interface. Most sh scripts can be run directly by Bash. All test cases in our experiment are shell scripts.

Our *bash* package was downloaded from the SIR site [18]. A total of 6 faulty programs were used, corresponding to the following *bash* versions: v2.01, v2.01.1, v2.02, v2.03, v2.04 and v2.05. We used the default faults already activated when the package was downloaded.

We designed 50 different if-then-else statements, 50 different loop statements, and 4 different select statements. Therefore, we obtained a total of $50 \times 50 \times 4 = 10,000$ test cases.

A total of 10 MRs are designed as follows:

i) MR1: "if (condition) {do A} else {do B}" is equivalent to "if (not(condition)) {do B} else {do A}."

ii) MR2: This MR converts "if" conditions to a different but equivalent form. For example, the following code:
a = 6
if (a $>$ 5)
should be equivalent to
a = 6
if (a $>$ 1 && a $>$ 5)

iii) MR3: This MR replaces a block of statements by a function. For example, "if (condition){block 1 of statements} else {block 2 of statements}" is equivalent to "if (condition){call function a} else {call function b}" where

functions a and b are equivalent to blocks 1 and 2 of statements, respectively.

iv) MR4: Similar to MR2. For example, the following code:
a = 1
if (a $>$ 5)
should be equivalent to:
a = 1
if (a $>$ 4 && a $>$ 5)
The difference between MR2 and MR4 is in their follow-up test cases: in MR2's follow-up test case, all conditions inside "if ()" will be evaluated; in MR4's follow-up test case, only the first condition inside "if ()" will be evaluated because of the left-to-right evaluation order and short-circuit evaluation of logical AND/OR expressions (for example, when evaluating a Boolean expression "A AND B," B will not be evaluated when A is false).

v) MR5: *bash* supports different loop constructs such as "for," "until" and "while" loops. MR5 converts one type of loop to another type by making use of the equivalence relation among them.

vi) MR6: In the initial test case of MR6, a Linux utility is called. In the follow-up test case, the utility is given an alias and it is the alias, rather than the utility's original name, that is called.

vii) MR7: The follow-up test case is constructed by applying a code obfuscator to the initial test case. The code obfuscator takes a shell script as input and generates an equivalent shell script that is hard for humans to read/understand. MR7 states that the original script and the obfuscated script should be equivalent and, hence, should produce the same outputs.

viii) MR8: In the initial test case of MR8, some system utilities are called that read the standard input. In the follow-up test case, these utilities are called by means of input redirection to a file. The expected relation is that the two executions should be equivalent (that is, they should produce the same output).

ix) MR9: This MR makes use of the equivalence of some predefined names and symbols. For example, $HOME is equivalent to the symbol $\sim$.

x) MR10: Inserting many blank spaces before a closing bracket will not affect the output of the shell script. For example:
"if (a $>$ 5) ..."
and
"if (a $>$ 5) ..."
should be equivalent.
We recognized that this MR might not be effective. The purpose of this research, however, is exactly to study the nature of effective and ineffective MRs.

C. Coverage monitoring

A test coverage program, namely, gcov, was used to collect coverage and frequency information of test case executions. The gcov tool is a utility to be used in concert with gcc. For all programs, we collected the statement and branch profiles for each execution on each single test case. Note that a metamorphic test involves two or more executions.

D. Experimental procedure

The aim of the experiments is to find the correlation (in terms of correlation coefficient r) between the distance measure and the failure-detection rate of MRs. For ease of presentation, let us use the subject program *spWiki* as an example to illustrate the experimental procedure for collecting r's. A total of 6 r's are collected for each faulty program since there are 6 different distance metrics. *spWiki* has 19 faulty versions, namely, $V_1, V_2, \ldots, V_{19}$, and 20 MRs, namely, $R_1, R_2, \ldots, R_{20}$. There is a set of 1,000 initial test cases, namely, $t_1, t_2, \ldots, t_{1000}$. All of the 20 MRs use the same set of initial test cases but different sets of follow-up test cases. The experimental procedure for *spWiki* is depicted in Figure 2. Treatments for other subject programs are similar.

Figure 2 shows that, for each faulty program V_i under test ($i = 1, 2, \ldots, 19$), the following 6 correlation coefficients are calculated: $r(SCMD)_i$ $r(SFHD)_i$, $r(SFMD)_i$, $r(BCMD)_i$, $r(BFHD)_i$ and $r(BFMD)_i$. All the experimental data (for all the 7 subject packages and all their faulty versions) followed an uncorrelated bivariate normal distribution, and Pearson's correlation was calculated.

An example of such an $r(BCMD)_i$, for certain faulty version V_i of *spWiki*, is shown in Figure 3. In this example $r(BCMD)_i = 0.927$, $p < 0.001$, which means that there is a significant strong positive correlation between the BCMD and the failure-detection rate of MRs for program V_i.

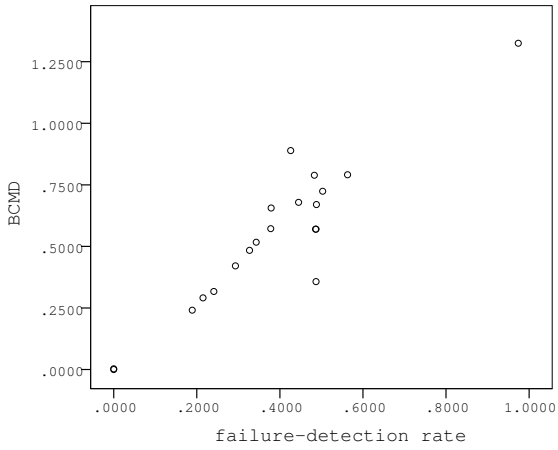


Fig. 3. Scatter plot for a faulty version of *spWiki*, where the x-axis represents the failure-detection rate, and the y-axis represents the mean Branch Coverage Manhattan Distance. Each point in the figure corresponds to an MR. There are 20 MRs (points), some of which overlap. The correlation coefficient $r = 0.927$, $p < 0.001$.

IV. Experimental Results

The experiments involved a total of 86 programs from 7 subject packages. Failures (in terms of violations of MRs) have been detected for 84 programs – there are 2 *spStudent* versions that did not reveal any failure. It is possible that these two programs are indeed correct; therefore, they are excluded from

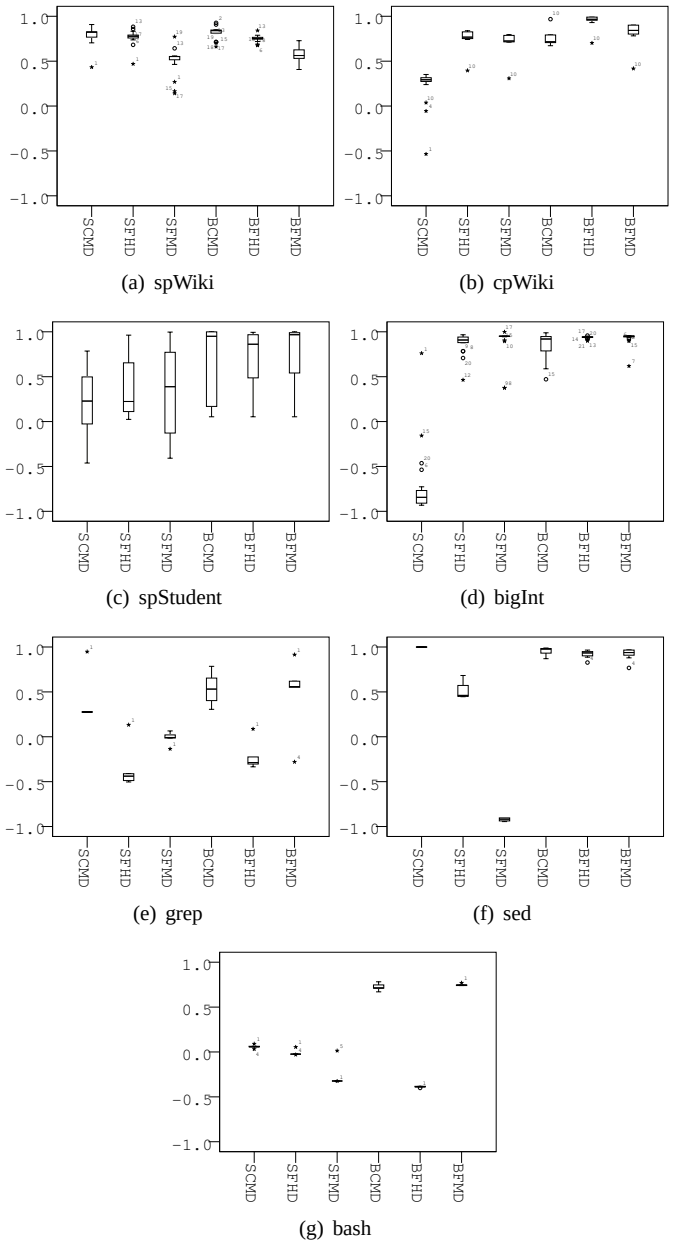
procedure EXPERIMENT
  Set *nbOfTestCases* to 1000;
  Set *nbOfVersions* to 19;
  Set *nbOfMRs* to 20;
  for each faulty version V_i **do**
    for each MR R_j **do**
      Set *nbOfFailures* to 0;
      for each initial test case t_k **do** /* The total number of initial test cases is *nbOfTestCases*. */
        Run $V_i(t_k)$ and get execution profiles EP_k of this run; /* EP_k includes branch and statement profiles */
        Generate a follow-up test case t'_k according to R_j, t_k and possibly the output of $V_i(t_k)$;
        Run $V_i(t'_k)$ and get execution profiles EP'_k of this run;
        if the current MR R_j is violated **then**
          Set *nbOfFailures* to *nbOfFailures* + 1;
        end if
        Calculate distances $SCMD_k$, $SFHD_k$, $SFMD_k$, $BCMD_k$, $BFHD_k$ and $BFMD_k$ using EP_k and EP'_k;
      end for
      /* Calculate the failure-detection rate and the 6 mean distances for V_i and R_j as follows: */
      Set $FailureDetectionRate_{i,j}$ to $nbOfFailures \div nbOfTestCases$;
      Set $\overline{SCMD}_{i,j}$ to $\left(\sum_{k=1}^{nbOfTestCases} SCMD_k\right) \div nbOfTestCases$;
      Set $\overline{SFHD}_{i,j}$ to $\left(\sum_{k=1}^{nbOfTestCases} SFHD_k\right) \div nbOfTestCases$;
      Set $\overline{SFMD}_{i,j}$ to $\left(\sum_{k=1}^{nbOfTestCases} SFMD_k\right) \div nbOfTestCases$;
      Set $\overline{BCMD}_{i,j}$ to $\left(\sum_{k=1}^{nbOfTestCases} BCMD_k\right) \div nbOfTestCases$;
      Set $\overline{BFHD}_{i,j}$ to $\left(\sum_{k=1}^{nbOfTestCases} BFHD_k\right) \div nbOfTestCases$;
      Set $\overline{BFMD}_{i,j}$ to $\left(\sum_{k=1}^{nbOfTestCases} BFMD_k\right) \div nbOfTestCases$;
    end for
    /* The following statement calculates $r(SCMD)_i$, which denotes the correlation coefficient between the failure-detection rate and the mean *SCMD* of MRs for faulty version V_i. */
    Calculate $r(SCMD)_i$ using the following points: $(FailureDetectionRate_{i,1}, \overline{SCMD}_{i,1})$, $(FailureDetectionRate_{i,2}, \overline{SCMD}_{i,2})$, ..., $(FailureDetectionRate_{i,nbOfMRs}, \overline{SCMD}_{i,nbOfMRs})$;
    Calculate $r(SFHD)_i$, $r(SFMD)_i$, $r(BCMD)_i$, $r(BFHD)_i$ and $r(BFMD)_i$ similarly.
  end for
end procedure

Fig. 2. Experimental procedure for collecting correlation coefficients between failure-detection rate and distance measure, using *spWiki* as an example

experiments. Further, one *bash* version had segmentation faults on many follow-up test cases during metamorphic testing and, as a result, execution profiles could not be collected. This version of *bash* was also excluded from experiments. This section, therefore, will report the experimental results of the remaining 83 faulty programs.

Mean results of correlation coefficients and p-values are summarized in Table II. Because a correlation coefficient (Pearson's r) greater than or equal to 0.50 indicates a strong correlation [19], cells in Table II(a) are highlighted if their values are 0.50 or higher, indicating that the corresponding distance metric is strongly correlated to fault-detection effectiveness. Several observations can be made. First, each of the 6 distance metrics has certain cells highlighted for certain subject programs; however, the branch-based metrics are obviously stronger than the statement-based metrics. Secondly, BCMD appears to be the best among the 6 metrics as all its cells are highlighted. The second best is BFMD, with one cell (for *grep*) lower than 0.50. The above findings are consistent with previous observations that branch-based metrics are often more effective than statement-based metrics and that coverage-based metrics are often more reliable than frequency-based metrics [13]. The distributions of correlation coefficients are shown in Figure 4.

Table II(b) summarizes mean p-values. Table cells whose values are 0.05 or lower are highlighted to indicate that the correlation is statistically significant. Similar observations can be made, that is, while each of the 6 distance metrics has certain cells highlighted for certain subject programs, the branch-based metrics are obviously more statistically significant than the statement-based metrics. Further, BCMD appears to be the best as all its cells are highlighted. The second best is BFMD,



(a) spWiki (b) cpWiki

(c) spStudent (d) bigInt

(e) grep (f) sed

(g) bash

Fig. 4. Distributions of correlation coefficients

with one cell (for *grep*) higher than 0.50.

It can be concluded, therefore, that there is a significant strong positive correlation between BCMD and the fault-detection effectiveness of MRs, and this correlation is reliable in the sense that it persists across all of the subject programs studied. This finding strongly answers the research question raised in Section I.

V. Comparison with related work

Asrafi et al. [20] suggested that "there is a certain degree of correlation between the code coverage achieved by a metamorphic relation and its fault-detection effectiveness." The code coverage they proposed is the "accumulative coverage

TABLE II
Mean correlation coefficients and p-values

programID	SCMD	SFHD	SFMD	BCMD	BFHD	BFMD
spWiki	0.79	0.77	0.50	0.82	0.75	0.58
cpWiki	0.22	0.76	0.72	0.75	0.96	0.83
spStudent	0.22	0.37	0.33	0.66	0.71	0.76
bigInt	-0.71	0.87	0.89	0.84	0.94	0.93
grep	0.41	-0.34	-0.01	0.54	-0.21	0.47
sed	1.00	0.52	-0.92	0.95	0.92	0.92
bash	0.06	-0.01	-0.26	0.72	-0.39	0.75
average	0.28	0.42	0.18	0.75	0.52	0.75

(a) mean correlation coefficient (Pearson's r)

programID	SCMD	SFHD	SFMD	BCMD	BFHD	BFMD
spWiki	0.00305	0.00200	0.07947	0.00011	0.00016	0.01437
cpWiki	0.00785	0.00155	0.00279	0.00127	0.00557	0.00346
spStudent	0.14378	0.27536	0.18523	0.00038	0.04329	0.03363
bigInt	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
grep	0.25580	0.19700	0.86480	0.04280	0.39380	0.07820
sed	0.00000	0.00314	0.00000	0.03929	0.03400	0.00000
bash	0.86420	0.93160	0.36220	0.02620	0.26460	0.01240
average	0.18210	0.20152	0.21350	0.01572	0.10592	0.02029

(b) mean p-value

percentages," which is achieved by accumulating the code coverage of both the initial and follow-up executions of an MR over the entire set of test cases. Two subject programs were used in their empirical study, namely TCAS (173 lines of C code) and KNASPSACK (780 lines of Java code).

We investigated the correlation coefficients between accumulative coverage percentage (for statement and branch coverages) and failure-detection rate of MRs using all the 7 subject packages. A strong correlation is found only for the small programs *bigInt* and *cpWiki*. For all the other programs, the mean correlation coefficients are all below 0.35. This finding suggests that the "distance" between initial and follow-up test cases proposed in the present paper should be more useful than "accumulative coverage," especially for larger programs.

In software testing literature there has been much work on test case selection and prioritization techniques using the concept of "similarity" of test cases [21]. The present research is different from these techniques as our objective is to reveal the nature of good MRs, rather than good individual test cases. It should be noted, however, that a failure-causing metamorphic test must involve at least one failure-causing test case (which could be the initial or the follow-up test case or both). How to integrate similarity-based MR selection and test case selection will be an important future research topic.

VI. Discussions and Conclusion

Metamorphic testing is a practical approach to alleviating the oracle problem. For a given problem, normally more than one MR can be identified. Because testing resources are always limited, it is important to know which MRs should be given priority for software testing.

In earlier work it was suggested that MRs whose initial and follow-up executions have larger dissimilarities may have higher chances of revealing failures. The concept of dissimilarity, however, was not clearly defined. In this research we proposed 6 metrics to measure the dissimilarity between initial and follow-up executions, and conducted a series of empirical studies to investigate the correlation between these metrics and the fault-detection effectiveness of MRs. It is found that the branch-based metrics have a stronger correlation and, in

particular, the BCMD metric constantly has a significant strong positive correlation with the fault-detection effectiveness of MRs across all of the subject programs. This finding gives an affirmative answer to the research question raised in Section I.

With regard to the internal validity of this work, all the code was carefully checked and the experimental data (both intermediate results and final results) were carefully reviewed. With regard to the conclusion validity, appropriate statistical methods were used and the findings have a strong statistical significance. With regard to the external validity, both small, medium to large, and large subject programs were used in the experiments, and both real and seeded faults were involved. The external validity can be enhanced by considering more types of MRs, especially non-identity relations, and by further empirical studies.

There are several possible ways to employ the findings of this research to conduct cost-effective metamorphic testing in practice. First, the developers of the software often have good knowledge of their algorithms and code and, therefore, may be able to estimate or guess which MRs could give a larger BCMD measure before running any test case. Secondly, following the idea of *software cybernetics* [22], feedback-based selection strategies can be developed to dynamically select MRs based on their coverage information collected online. We have developed such a framework with encouraging preliminary empirical evaluation results. This framework will be reported in the near future. Thirdly, in the context of regression testing and observation-based testing, test case coverage data is available (either for the previous versions or for the current version of the program under test). In these situations MRs can be selected or prioritized directly using the available coverage data. In addition to MRs, the pairs (or tuples) of initial and follow-up test cases may also be prioritized by globally considering the most dissimilar pairs (or tuples) of test cases. It is also possible to make use of the findings of this research to help with software reliability estimation. For example, after a system has passed metamorphic testing conducted by users, the initial and follow-up execution distance data will become available. If the distances are large then higher confidence could be established in the system's reliability because the MRs are expected to be effective in detecting failures. Future research on this topic will involve extensive empirical studies.

Acknowledgments

We are grateful to Dehao Huang for his discussions on this work. This research was supported in part by a linkage grant of the Australian Research Council (project ID: LP100200208).

References

[1] L. I. Manolache and D. G. Kourie, "Software testing using model programs," *Software: Practice and Experience*, vol. 31, no. 13, pp. 1211–1236, 2001.

[2] T. Y. Chen, T. H. Tse, and Z. Q. Zhou, "Semi-proving: An integrated method based on global symbolic evaluation and metamorphic testing," in *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'02)*. ACM Press, New York, 2002, pp. 191–195.

[3] T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Q. Zhou, "Metamorphic testing: Applications and integration with other methods," in *Proceedings of the 12th International Conference on Quality Software (QSIC'12)*. IEEE Computer Society Press, 2012, pp. 285–288.

[4] M. Blum and S. Kannan, "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, pp. 86–97.

[5] R. J. Lipton, "New directions in testing," in *Proceedings of the DIMACS Workshop on Distributed Computing and Cryptography*. American Mathematical Society, Providence, RI, 1991, pp. 191–202.

[6] M. Blum, M. Luby, and R. Rubinfeld, "Self-testing / correcting with applications to numerical problems," *Journal of Computer and System Sciences*, vol. 47, no. 3, pp. 549–595, 1993.

[7] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. H. Tse, F.-C. Kuo, and T. Y. Chen, "Automated functional testing of online search services," *Software Testing, Verification and Reliability*, vol. 22, no. 4, pp. 221–243, 2012.

[8] T. Y. Chen, T. H. Tse, and Z. Q. Zhou, "Semi-proving: An integrated method for program proving, testing, and debugging," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 109–125, 2011.

[9] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu, "Metamorphic slice: an application in spectrum-based fault localization," *Information and Software Technology*, vol. 55, no. 5, pp. 866–879, 2013.

[10] T. Y. Chen, T. H. Tse, and Z. Q. Zhou, "Fault-based testing without the need of oracles," *Information and Software Technology*, vol. 45, no. 1, pp. 1–9, 2003.

[11] 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 *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC'04)*. Madrid, Spain: Polytechnic University of Madrid, 2004, pp. 569–583.

[12] Z. Q. Zhou, "Using coverage information to guide test case selection in adaptive random testing," in *Proceedings of the 34th Annual International Computer Software and Applications Conference (COMPSAC'10), 7th International Workshop on Software Cybernetics*. IEEE Computer Society Press, 2010, pp. 208–213.

[13] Z. Q. Zhou, A. Sinaga, and W. Susilo, "On the fault-detection capabilities of adaptive random test case prioritization: Case studies with large test suites," in *Proceedings of the 45th Annual Hawaii International Conference on System Sciences (HICSS'12)*. IEEE Computer Society Press, 2012, pp. 5584–5593.

[14] W. Dickinson, D. Leon, and A. Podgurski, "Finding failures by cluster analysis of execution profiles," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE'01)*. IEEE Computer Society Press, 2001, pp. 339–348.

[15] W. Masri, A. Podgurski, and D. Leon, "An empirical study of test case filtering techniques based on exercising information flows," *IEEE Transactions on Software Engineering*, vol. 33, no. 7, pp. 454–477, 2007.

[16] G. Rothermel, R. H. Untch, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[17] S. Yoo and M. Harman, "Regression testing minimization, selection and prioritization: a survey," *Software Testing, Verification and Reliability*, vol. 22, pp. 67–120, 2012.

[18] H. Do, S. G. Elbaum, and G. Rothermel, "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact," *Empirical Software Engineering: An International Journal*, vol. 10, no. 4, pp. 405–435, 2005.

[19] A. Rubin, *Statistics for Evidence-Based Practice and Evaluation*, 2nd ed. Brooks/Cole, 2010.

[20] M. Asrafi, H. Liu, and F.-C. Kuo, "On testing effectiveness of metamorphic relations: A case study," in *Proceedings of the 5th International Conference on Secure Software Integration and Reliability Improvemen (SSIRI'11)*. IEEE Computer Society Press, 2011, pp. 147–156.

[21] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology*, vol. 22, no. 1, pp. 6:1–6:42, 2013.

[22] K.-Y. Cai, J. W. Cangussu, R. A. DeCarlo, and A. P. Mathur, "An overview of software cybernetics," in *Proceedings of the 11th International Workshop on Software Technology and Engineering Practice*. IEEE Computer Society Press, 2003, pp. 77–86.