Metamorphic Testing: A Review of Challenges and Opportunities

TSONG YUEH CHEN and FEI-CHING KUO, Swinburne University of Technology
HUAI LIU, Victoria University
PAK-LOK POON, RMIT University
DAVE TOWEY, University of Nottingham Ningbo China
T. H. TSE, The University of Hong Kong
ZHI QUAN ZHOU, University of Wollongong

Metamorphic testing is an approach to both test case generation and test result verification. A central element is a set of metamorphic relations, which are necessary properties of the target function or algorithm in relation to multiple inputs and their expected outputs. Since its first publication, we have witnessed a rapidly increasing body of work examining metamorphic testing from various perspectives, including metamorphic relation identification, test case generation, integration with other software engineering techniques, and the validation and evaluation of software systems. In this paper, we review the current research of metamorphic testing and discuss the challenges yet to be addressed. We also present visions for further improvement of metamorphic testing and highlight opportunities for new research.

CCS Concepts: •**Software and its engineering → Software verification and validation; Software testing and debugging;**

Additional Key Words and Phrases: Metamorphic testing, metamorphic relation, test case generation, oracle problem

ACM Reference format:
Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Pak-Lok Poon, Dave Towey, T. H. Tse, and Zhi Quan Zhou. 2017. Metamorphic Testing: A Review of Challenges and Opportunities. *ACM Comput. Surv.* 000, 000, Article 000 (September 2017), 27 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

This research was supported in part by a linkage grant of the Australian Research Council (project ID LP100200208) and a grant of the General Research Fund of the Research Grants Council of Hong Kong (project no. 716612). Dave Towey acknowledges the financial support from the Artificial Intelligence and Optimisation Research Group of the University of Nottingham Ningbo China, the International Doctoral Innovation Centre, the Ningbo Education Bureau, the Ningbo Science and Technology Bureau, and the University of Nottingham.
Author's addresses: T. Y. Chen and F.-C. Kuo, Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn VIC 3122, Australia. H. Liu, College of Engineering & Science, Victoria University, Melbourne VIC 8001, Australia. P.-L. Poon, School of Business IT and Logistics, RMIT University, Melbourne, VIC 3001, Australia. D. Towey, School of Computer Science, University of Nottingham Ningbo China, Ningbo, Zhejiang 315100, China. T. H. Tse, Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Z. Q. Zhou, Institute of Cybersecurity and Cryptology, School of Computing and Information Technology, University of Wollongong, Wollongong, NSW 2522, Australia.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2017 Copyright held by the owner/author(s). Publication rights licensed to ACM. 0360-0300/2017/9-ART000 $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Software testing is a mainstream approach to software quality assurance and verification. However, it faces two fundamental problems: the oracle problem and the reliable test set problem. The *oracle problem* refers to situations where it is extremely difficult, or impossible, to verify the test result of a given *test case* (that is, an input selected to test the program). Normally, after the execution of a test case t, a systematic mechanism called a *test oracle* (or simply an *oracle*) is required to check the execution result. If the result does not agree with the expected outcome, we say that t fails and refer to it as a *failure-causing test case*. Otherwise, we say that t succeeds and refer to it as a *successful*, or *non-failure-causing*, *test case*. In many real-life situations, however, an oracle may not exist, or it may exist but resource constraints make it infeasible to use. The *reliable test set problem* means that since it is normally not possible to exhaustively execute all possible test cases, it is challenging to effectively select a subset of test cases (the reliable test set) with the ability to determine the correctness of the program. A number of strategies have been proposed to generate test cases for addressing the reliable test set problem, including random testing [39], coverage testing [101], search-based testing [40], and symbolic execution [11]. Compared with test case generation strategies, only a few techniques have been proposed to address the oracle problem, such as assertion checking [79] and N-version programming [61]. When the oracle problem occurs, many strategies for the reliable test set problem have limited applicability and effectiveness. Regardless of how effective a strategy is in generating a failure-causing test case, unless it leads to a crash of the program under test, that failure may not be recognized in the presence of the oracle problem.

Unlike most other software testing techniques, *Metamorphic Testing* (*MT*) [15] can be used for both test case generation and test result verification — thus addressing both fundamental problems of testing. Although it was initially proposed as a method to generate new test cases based on successful ones, it soon became apparent that MT is also an effective approach for alleviating the oracle problem. A central element of MT is a set of *Metamorphic Relations* (*MRs*), which are necessary properties of the target function or algorithm in relation to multiple inputs and their expected outputs. When implementing MT, some program inputs (called *source inputs*) are first generated as *source test cases*, on the basis of which an MR can then be used to generate new inputs as *follow-up test cases*. Unlike the traditional way of verifying the test result of each individual test case, MT verifies the source and follow-up test cases as well as their outputs against the corresponding MR.

Since its first publication in 1998, quite a number of studies have been conducted on various aspects of MT. In recent years especially, MT has been attracting an increasing amount of attention and has helped detect a large number of real-life faults. It was a surprise to the software testing community, for example, when MT managed to detect new faults [77, 93] in three out of seven programs in the Siemens suite [43] even though these programs had repeatedly been studied in major software testing research projects for 20 years. In addition, Le et al. [50] detected over one hundred faults in two popular C compilers (GCC and LLVM) based on a simple relation, which was quickly realized to be an MR [51, 78]. In addition to its extensive use in software testing [8, 13, 14, 16, 20, 50–52, 83, 91, 99], MT has been widely applied to address the oracle problem in the broader context of software engineering [3, 27, 29, 45, 46, 57, 77, 92, 93]. It has also been used as a technique for validation [91] and quality assessment [98], detecting real-life faults in several popular search engines.

In recent surveys of the oracle problem [4, 68, 71, 72], a significant amount of discussion was devoted to MT, which was categorized as a mainstream and promising approach for addressing the problem. Among the surveys specifically about MT [34, 42, 81], Segura et al. [81] have presented an extensive literature review of MT, analyzing and summarizing 119 research papers published between 1998 and 2015, highlighting some open questions. Among all the papers on MT, we consider some of them as the *most important and influential* studies if either they opened new and important research directions for MT or their results have had significant impact. For example, some studies presented various approaches to systemically generate metamorphic relations [25, 47, 96, 100]. Other studies proposed the innovative application of MT to, amongst others, proving [27, 37], debugging [29, 46], fault localization [92], fault tolerance [57], and program repair [45]. Still other studies, as discussed above, have had the surprising and striking results of detecting real-life bugs in, amongst others, popular compilers [50, 51] and search engines [98]. Our present article is different from traditional surveys. Rather than providing an exhaustive survey of what has been investigated, we focus instead on the above-mentioned most important and influential MT studies, the relationships among them, and their impacts. Complementary to previous surveys on the oracle problem [4, 68, 71, 72] and MT [34, 42, 81], we attempt to summarize and analyze results based on related studies from a different perspective, providing an in-depth discussion of what has really been achieved and what still remains to be done. The main contribution of this paper is threefold. To the best of our knowledge:

- It provides by far the most thorough summary and clarification of the critical concepts of MT, including improved formal notation and definitions as well as consolidated advantages of MT (Section 2); and important, but frequently overlooked or misunderstood concepts in MT (Section 3).
- It presents by far the most systematic discussions of MT's research in the contexts of (i) traditional software testing (Section 4); (ii) extension beyond testing, such as for proving MRs and for validation (Section 5); and (iii) integration of MT with other software engineering methods to address the oracle problem in related fields (Section 6). In each discussion, we first provide a high-level review of the state of the art of MT, and then highlight the critical challenges to be addressed.
- It unveils by far the most comprehensive list of contemporary opportunities for emerging research related to MT (Section 7).

2 BACKGROUND

Before formally presenting the notation and definitions, we first introduce the history of how MT was proposed, paving a way for a deeper understanding of its underlying intuitions, and facilitating the presentation of its evolution.

2.1 Are successful test cases really useless?

As pointed out by Dijkstra [31], software testing can only demonstrate the presence of faults, not their absence. In many situations, successful (non-failure-causing) test cases had been regarded as useless — because they do not reveal failures — and their test results were usually not passed to the debugging team. About twenty years ago, we revisited the question: *Are successful test cases really useless?* Our answer was "no". Most test case generation strategies serve specific purposes, so every generated test case should carry some useful information about the program under test [15, 21]. It has been an interesting (and challenging) task to examine how to make use of such useful, but implicit, information to support further testing.

Our revisit of this question led to the development of metamorphic testing (MT). In MT, we first identify some necessary properties of the target function or algorithm in the form of metamorphic relations (MRs) among multiple inputs and their expected outputs. These MRs are then used to transform existing (source) test cases into new (follow-up) test cases. Obviously, because the follow-up test cases depend on the source test cases, they should also possess some (if not all) of the useful information embedded in them. If the actual outputs of source and follow-up test cases violate a certain MR, then we can say that the program under test is faulty with respect to the property associated with that MR. Although MT was initially proposed as a method for generating new test cases based on successful ones, it soon became clear that it could be used regardless of whether the source test cases were successful or not. In addition, it actually provided a lightweight, but effective, mechanism for test result verification — MT was thus recognized as a promising approach for alleviating the oracle problem.

It should be noted that MT is not the only technique designed to make use of successful test cases. *Adaptive Random Testing* (ART) [21, 26, 60] attempts to evenly spread the test cases across the input domain, using the location of successful test cases to guide selection of subsequent ones.

2.2 The intuition and formalization of MT

The intuition behind using MT to alleviate the oracle problem is as follows: Even if we cannot determine the correctness of the actual output for an individual input, it may still be possible to use relations among the expected outputs of multiple related inputs (and the inputs themselves) to help. Consider the following example.

Example 1. *Suppose that an algorithm f computes the shortest path for an undirected graph G, and a program P implements f. For any two vertices a and b in a large G, it may be very difficult to verify whether $P(G, a, b)$ — the computed result of P given the inputs G, a, and b — is really the shortest path between a and b. One possible way to verify the result is to generate all possible paths from a to b, and then check against them whether $P(G, a, b)$ is really the shortest path. However, it may not be practically feasible to generate all possible paths from a to b as the number of possible paths grows exponentially with the number of vertices. Although the oracle problem exists for testing the program P, we can make use of some properties to partially verify the result. For example, an MR can be derived from the following property: If the vertices a and b are swapped, the length of the shortest path will remain unchanged, that is, $|f(G, b, a)| = |f(G, a, b)|$. Based on this MR, we need two test executions, one with the source test case (G, a, b) and the other with the follow-up test case (G, b, a). Instead of verifying the result of a single test execution, we verify the results of the multiple executions against the MR — we check whether the relation $|P(G, b, a)| = |P(G, a, b)|$ (where we simply replace f by P) is satisfied or violated. If a violation is detected, we can then say that P is faulty.*

The following is a formal presentation of the MT methodology.

Definition 1 (Metamorphic Relation (MR)). *Let f be a target function or algorithm. A **metamorphic relation (MR)** is a necessary property[1] of f over a sequence of two or more inputs $\langle x_1, x_2, \ldots, x_n\rangle$, where $n \geqslant 2$, and their corresponding outputs $\langle f(x_1), f(x_2), \ldots, f(x_n)\rangle$. It can be expressed as a relation $\mathcal{R} \subseteq X^n \times Y^n$, where $\subseteq$ denotes the subset relation, and X^n and Y^n are the Cartesian products of n input and n output spaces, respectively. Following standard informal practice, we may simply write $\mathcal{R}\big(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)\big)$ to indicate that $\langle x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)\rangle \in \mathcal{R}$.*

For ease of presentation, we will write "target function or algorithm" as "target algorithm" in the remaining part of this paper.

For instance, the property from Example 1, "If the vertices a and b are swapped, the length of the shortest path will remain unchanged", is a necessary property of the target algorithm f. $|f(G, b, a)| = |f(G, a, b)|$ is the MR corresponding to this property.

[1]A necessary property of an algorithm means a condition that can be logically deduced from the algorithm.

Definition 2 (Source Input and Follow-up Input). *Consider an MR $\mathcal{R}\big(x_1,\ x_2,\ \ldots,\ x_n,\ f(x_1),\ f(x_2),\ \ldots,\ f(x_n)\big)$. Suppose that each x_j $(j = k+1, k+2, \ldots, n)$ is constructed based on $\langle x_1,\ x_2,\ \ldots,\ x_k,\ f(x_1),\ f(x_2),\ \ldots,\ f(x_k)\rangle$ according to $\mathcal{R}$. For any $i = 1, 2, \ldots, k$, we refer to x_i as a* ***source input****. For any $j = k+1, k+2, \ldots, n$, we refer to x_j as a* ***follow-up input****. In other words, for a given $\mathcal{R}$, if all source inputs x_i $(i = 1, 2, \ldots, k)$ are specified, then follow-up inputs x_j $(j = k+1, k+2, \ldots, n)$ can be constructed based on the source inputs and, if necessary, their corresponding outputs.*

In Example 1, (G, a, b) is the source input and (G, b, a) is the follow-up input constructed by using the same graph G and swapping the start and end nodes (a and b). Obviously, (G, a, b) and (G, b, a) can be used as test cases for MT (and are referred to as the source and follow-up test cases, respectively).

Definition 3 (Metamorphic Group of Inputs (MG)). *Consider an MR $\mathcal{R}\big(x_1,\ x_2,\ \ldots,\ x_n,\ f(x_1),\ f(x_2),\ \ldots,\ f(x_n)\big)$. The sequence of inputs $\langle x_1,\ x_2,\ \ldots,\ x_n\rangle$ is defined as a* ***metamorphic group (MG)*** *of inputs for the MR. More specifically, the MG is the sequence of source inputs $\langle x_1,\ x_2,\ \ldots,\ x_k\rangle$ and follow-up inputs $\langle x_{k+1},\ x_{k+2},\ \ldots,\ x_n\rangle$ related to $\mathcal{R}$.*

In Example 1, $\langle (G, a, b), (G, b, a)\rangle$ is an MG.

Definition 4 (Metamorphic Testing (MT)). *Let P be an implementation of a target algorithm f. For an MR $\mathcal{R}$, suppose that we have $\mathcal{R}\big(x_1,\ x_2,\ \ldots,\ x_n,\ f(x_1),\ f(x_2),\ \ldots,\ f(x_n)\big)$.* ***Metamorphic testing (MT)*** *based on this MR for P involves the following steps:*

(1) Define $\mathcal{R}'$ by replacing f by P in $\mathcal{R}$.

(2) Given a sequence of source test cases $\langle x_1,\ x_2,\ \ldots,\ x_k\rangle$, execute them to obtain their respective outputs $\langle P(x_1),\ P(x_2),\ \ldots,\ P(x_k)\rangle$. Construct and execute a sequence of follow-up test cases $\langle x_{k+1},\ x_{k+2},\ \ldots,\ x_n\rangle$ according to $\mathcal{R}'$ and obtain their respective outputs $\langle P(x_{k+1}),\ P(x_{k+2}),\ \ldots,\ P(x_n)\rangle$.

(3) Examine the results with reference to $\mathcal{R}'$. If $\mathcal{R}'$ is not satisfied, then this MR has revealed that P is faulty.

If conducting MT for Example 1, f would first be replaced by P in the MR to give the expected relation $|P(G, b, a)| = |P(G, a, b)|$. Given the MG $\langle (G, a, b), (G, b, a)\rangle$, the program would then be executed so that we could examine whether $|P(G, b, a)| = |P(G, a, b)|$ is satisfied or violated.

With MT, it is not necessary to investigate whether $P(x_i) = f(x_i)$ for any individual test case x_i — which would require a test oracle. MT therefore alleviates the oracle problem in testing.

2.3 Advantages of MT

Based on the definitions in the previous section, we next summarize MT's main advantages. Note that although these advantages are not unique to metamorphic testing, MT is one of the few techniques that have all of them.

Advantage 1: Simplicity in concept. Both the intuition and technical content of MT are simple and elegant. As shown in previous studies [55, 73], testers, even those without much experience or expertise, could learn how to use MT in a few hours and then correctly apply it to test a variety of systems.

Advantage 2: Straightforward implementation. According to Definition 4 (Section 2.2), implementing MT is straightforward. Both test case generation and test result verification are implemented based on MRs. Previous studies of MT, especially those related to MT applications where a large number of MRs are identified, suggest that MR identification is not a very difficult task even though it cannot be completely automated. The success of using a very simple MR to detect hundreds of real-life bugs in two popular compilers [50] is strong evidence that identification of good MRs may not be difficult at all. It should also be straightforward for users to develop MT tools for their own specific domains [84, 100].

Advantage 3: Ease of automation given the availability of MRs. Apart from the MR identification process, it should not be difficult to automate the major steps in MT, including test case generation, execution, and verification. The construction of individual test cases is simple. Source test cases can be generated through existing testing methods while follow-up test cases can be constructed through transformations according to MRs. Test case execution is normally straightforward also, and thus may be the most easily automated process for almost all testing methods. Test result verification in MT can also be automated by creating scripts to check test outputs against the relevant MRs. In the entire MT procedure, the only part that might not be fully automated is MR identification, but this can be improved based on the recent influential study of systematic MR identification [25]. Although tools already exist that implement the entire MT process for certain application domains [84, 100], further research is still required to develop a general framework incorporating and automating every MT step as much as possible.

Advantage 4: Low costs. Compared with traditional testing techniques, MT requires a process of identifying MRs, and incurs marginal additional computational costs for generation and execution of follow-up test cases, and test result verification. Although MR identification involves some manual work and hence incurs some overheads, it is expectable and unavoidable. Similar manual processes are necessary in traditional testing, such as the requirements analysis for specification-based testing, the construction of formal models for model-based testing, the identification of assertions, and the design of fitness functions for search-based testing. As explained above, follow-up test cases are easily generated through transformations according to MRs, and usually incur very low cost. Although test result verification involves checking outputs against MRs, the associated overhead is relatively low compared with the cost of result verification when the oracle problem exists.

Another important factor affecting costs is the scalability problem, by which we mean that the required number of test cases or required testing efforts is exponentially growing with the size of the program under test. For example, the multiple-condition coverage criterion, widely regarded as "one of the most popular criteria in software testing practice" [101], aims to design test cases that cover all possible combinations of condition outcomes in a decision for a given program. For a given program, such a testing criterion requests a minimum number of test cases to satisfy its original objective. In contrast, there exist several techniques, such as MT and random testing, that do not have this kind of constraint on the minimum number of test cases. MT can be applied with *a test suite of any size*, independent from the size and complexity of the program under test. How large or small the test suite is does not affect the implementation of MT. Thus, MT does not have the scalability problem as encountered by the multiple-condition test case selection method. Some may argue that MT may require many test cases to guarantee the detection of certain software failures. However, this issue is related to the failure rate of the program under test and fault-detection capability rather than scalability.

3 FREQUENTLY MISUNDERSTOOD CONCEPTS IN MT

In our research, we have identified the following MT concepts that were frequently overlooked or misunderstood — they appear to be the cause of most inquiries from readers, reviewers, and software practitioners. In this section, therefore, we highlight and address each one of them, providing a more comprehensive picture of MT and thus enabling a deeper understanding of MT's capabilities.

Concept 1: Not all necessary properties are MRs. MRs are necessary properties of the target algorithm in relation to multiple inputs and their corresponding expected outputs. Not all necessary properties of the algorithm, therefore, are MRs. For example, although $-1 \leq \sin(x) \leq 1$ is a necessary property of the sine function, it only involves a single instance of the input and thus cannot be considered an MR — even though, obviously, violation of this property implies that the relevant program is faulty. It should be noted that such a property involving only one input has been used in other techniques, such as assertion checking [79], which also addresses the oracle problem but in a different way and less effective than MT in detecting various faults [41, 97]. There are also development and testing approaches that involve multiple executions using the same input: N-version programming [61] and differential testing [35], for instance, verify the test results against the property that various versions of the same software should produce the same results given the same input. However, because such a property does not involve multiple different inputs (even though it involves multiple executions across various versions), it is not regarded as an MR.

Concept 2: Not all MRs separate into input-only and output-only sub-relations. Many previously studied MRs consist of two separate or independent components: one sub-relation involving only the inputs and the other one involving only the outputs. Consider the shortest path program $P(G, a, b)$ in Example 1 (Section 2.2), where G is an undirected graph, a is the start node, b is the end node, and the output is a shortest path from a to b. The given MR, "If the vertices a and b are swapped, the length of the shortest path will remain unchanged", which we denote as MR_1, can be decomposed into two separate sub-relations: R_{in} (a relation only involving the inputs: "the start and end nodes, a and b, are swapped") and R_{out} (a relation only involving the outputs: "$|P(G, b, a)| = |P(G, a, b)|$"). Although this type of MR is often identified, it should be noted that there are other types that cannot be decomposed into input-only and output-only sub-relations. Consider a second MR, denoted as MR_2: $|P(G, a, c)| + |P(G, c, b)| = |P(G, a, b)|$ where c is a node appearing in the shortest path from a to b in graph G. In MR_2, the follow-up test cases (G, a, c) and (G, c, b) depend on the output of the source test case (G, a, b). Although MR_2 is different from MR_1, which is in the form of R_{in} and R_{out}, MR_2 does still comply with Definition 1.

Concept 3: Not all MRs are equality relations. Although many of the MRs studied to date have involved equality relations, this is not a requirement in the original MR definition (Definition 1). Consider the following example.

Example 2. *Suppose that a database query command q extracts data from the database using the condition $c_1 \vee c_2 \vee \ldots \vee c_n$. One possible MR for q is: If any c_i $(1 \leq i \leq n)$ is removed, the new extracted data should be a subset of the original extracted data, that is, $q(c_1 \vee c_2 \vee \ldots \vee c_{i-1} \vee c_{i+1} \vee \ldots \vee c_n) \subseteq q(c_1 \vee c_2 \vee \ldots \vee c_n)$.*

Unlike the MR defined in Example 1, which involves an equality relation, the MR in Example 2 involves a relation that is not an equality. Furthermore, some studies have considered the use of nondeterministic or probabilistic relations as a kind of extension to MT [38, 66]. In fact, the MR definition was never constrained to specific relation types. Although other techniques (such as the data diversity approach for fault tolerance [2]) specifically involve the use of equality relations, MRs *may* include but are not limited to equality relations. This makes MT intrinsically different from other techniques. Interested readers who wish to further explore the differences between MT and these other techniques may consult our previous studies [29, 55].

Concept 4: MT can be applied with or without an oracle. Although MT has been extensively applied to the testing of programs with the oracle problem, it can also be applied when a usable oracle is available — something that has been overlooked or misunderstood by many researchers. In fact, MT has revealed real-life faults that had remained undetected for years in some small-sized and extensively-tested programs — such as the famous Siemens programs [77, 93], which will be discussed in Section 4.2. This means that MT can be used as a test case generation strategy regardless of whether or not a usable test oracle exists. As will be shown in Section 5.1, MT with semi-proving can reveal conditions of inputs that lead to violations of an MR (if such violations exist). These conditions are useful for debugging, regardless of whether the test oracle exists or not. In summary, MT is a useful and effective method even when a test oracle exists.

4 MT IN TESTING

4.1 MT as an approach to alleviating the oracle problem

State of the art. Although it has widely been acknowledged that MT can effectively alleviate the oracle problem in testing, one can never completely solve it. MRs are necessary properties of the target algorithm in relation to multiple inputs and their expected outputs, but because there are usually a huge number of these properties, it is almost impossible to obtain a complete set of MRs representing all of them. Even if it were possible to obtain such a complete set of MRs, they might still not be equivalent to a test oracle due to the *necessary* (but *not sufficient*) nature of the properties. Nevertheless, a recent empirical study [55] has delivered very encouraging results, demonstrating how a small number of diverse MRs appear to be very close to the test oracle in terms of software fault-detection ability. For each of the six subject programs in the study, MT only required an average of three to six diverse MRs to reveal at least 90% of the faults that could be detected by an oracle.

The effectiveness of MT in alleviating the oracle problem has been shown repeatedly in numerous studies, covering many different domains, including bioinformatics [16, 75, 76, 80]; web services [14, 84]; embedded systems [13, 44, 49]; components [8, 59]; compilers [50, 86]; databases [52]; machine learning classifiers [65, 67, 91]; online search functions and search engines [98, 99]; software product lines [82, 83]; and security [20]. In particular, MT has detected real-life faults in some frequently used programs with the oracle problem. For example, when testing a program analyzing gene regulatory networks, Chen et al. [16] identified some MRs involving simply altering the basic network structure (through deletion of a node, or addition of an edge, for instance). It was surprising to observe that a fault was revealed by a very simple MR that added a zero-weight edge. Other examples of MT detecting real-life faults include its application in embedded systems [49], in three famous Siemens programs [77, 93], and with two popular C compilers [50, 86]. Readers can refer to the recent survey [81] for the details on how MT has addressed the oracle problems in these different application domains. We will not repeat the detailed discussions in this paper. Instead, we will now summarize the similarity among these studies. Most studies have used random testing (RT) as the benchmark for evaluating the fault-detection effectiveness of MT, either explicitly or implicitly. Usually, faults are seeded into a base program (automatically and/or manually) to generate a set of faulty versions called mutants. The base program can then be used as the oracle. RT with the oracle provides the upper bound of the fault-detection effectiveness. RT without the oracle provides the lower bound — in which case we can only detect faults related to program crashes, such as segmentation faults. It is usually reported that MT always detects more faults than RT without the oracle. Obviously, any program crash will lead to the violation of MRs, whereas some faults do not necessarily trigger crashes but may result in MR violations. On the other hand, the more MRs used, the closer will be the number of faults detected by MT to that for RT with the oracle. Furthermore, if a sufficient number of diverse MRs are used, then the fault-detection effectiveness of MT is found to approach that of the oracle [55].

In addition to RT, MT has also been compared with another technique for addressing the oracle problem, assertion checking [41, 97]. Although incurring a slightly higher computational cost, MT has been observed to detect more faults than assertion checking. MT was also found to be complementary to error trapping, a commonly used spreadsheet testing technique [74]. Because MT and error trapping find different types of faults, it has been proposed that they should both be used when testing spreadsheets.

Challenge 1: Comprehensive empirical studies for a unified understanding of MT. The increasing number of real-world programs tested using MT is indicative of its wide acceptance [81], but a thorough evaluation of MT's overall effectiveness is still lacking. Many experimental studies [12, 55] have used mutation analysis to evaluate the fault-detection effectiveness of MT — evaluating how well MT, or more specifically a set of MRs, can alleviate the oracle problem based on how many mutants can be killed. However, most of these studies either focused on one particular application domain, or were based on a set of small or medium-sized subject programs. In addition to the appropriate effectiveness measurement (such as the mutation-based metrics), further empirical studies involving large and complex subject programs, from a variety of application domains, will be needed to develop a full picture of MT's fault-detection effectiveness. Such projects are labor-intensive and time-consuming, and should be conducted through collaborations across different research groups with complementary strengths. Furthermore, some previous MT experiments have yielded contradictory results. As discussed in a previous survey [81], for instance, the effectiveness of MRs (such as those in [17, 62]) has not been conclusively determined. It is therefore critical that all these experiments be summarized and analyzed. Based on these analyses, more comprehensive and thorough empirical studies should be designed and conducted. It is hoped that such comprehensive studies will lead to a more unified understanding of MT, enabling provision of clearer directions and guidelines for further MT research.

Challenge 2: Systematic MR identification and selection. Effective MRs are the key to MT alleviating the oracle problem. Although many MRs have been identified for various application domains (as mentioned in Section 2.3), and were reportedly not difficult to identify, most of these identifications were conducted in an ad hoc and arbitrary way. Several studies have been conducted examining how to *systematically* identify MRs [25, 56] and how to select "good" MRs [12, 17, 55, 62] (the results of these studies have been summarized by Segura et al. [81]). However, both systematic identification and selection of appropriate MRs still face several critical challenges.

Research into MR identification is important, but still at a preliminary stage, with most techniques proposed so far having limited applicability. Currently, MR identification strategies can be classified as either ad hoc or systematic, with most previous MT studies requiring testers to identify MRs in an ad hoc manner, without any systematic mechanism. Recently, however, research has been emerging on systematic methods for MR identification. Zhang et al. [96], for example, proposed identifying MRs from multiple executions of the program under test. On the one hand, MRs based on program executions may be erroneous if the implementation is faulty. The latter is exactly what we set out to test. On the other hand, even though these MRs may not be valid, they provide users with clues and inspirations for identifying appropriate MRs.

Some MR identification techniques can only be applied in specific application domains [47, 96, 100], or may require the existence of initial MRs [33, 56]. Other recent work, however, has yielded much higher applicability and MR identification without a need for existing MRs: METRIC [25], for instance, is based on the concepts of category and choice [24] from the software specifications. Categories refer to input parameters or environmental conditions that affect the execution of the software under test, while choices are disjoint partitions of each category that cover sets of possible values for the category. Technically speaking, MRs are part of the specifications, and hence, intuitively, should be identifiable from them — an intuition that motivated the technique on which METRIC is based. However, in spite of its systematic approach, METRIC still relies somewhat on the testers' expertise and experience in identifying MRs. Consider the following example.

Example 3. *For the sine function, its specification is usually given by the equation*

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \cdots \tag{1}$$

With this specification, it may not be difficult to identify the MR "$\sin(-x) = -\sin(x)$". However, it is not trivial to derive some other MRs from Equation (1), such as "$\sin(\pi - x) = \sin(x)$" or "$\sin(x + 2\pi) = \sin(x)$", because the final cancellation of all π's after replacing x by $(\pi - x)$ or $(x + 2\pi)$ in the equation involves difficult mathematical operations.

On the other hand, if the sine function is defined to be the ratio of the opposite side to the hypotenuse in a right-angled triangle, then the MRs "$\sin(\pi - x) = \sin(x)$" and "$\sin(x + 2\pi) = \sin(x)$" should be easily identifiable — because they follow directly from the definition or specification.

Current practices in specifications engineering aim mainly to help developers understand the required functionality of the software to be developed, with a view to delivering a system that satisfies user needs. It will therefore be a challenge to investigate new specification practices that would support the identification of MRs. This research direction may bring a new perspective to specifications engineering.

Although work has been conducted into providing guidelines for "good" MR selection [12, 17, 55, 62], these guidelines remain rather qualitative and their implementation is still a relatively subjective process. More work is needed to produce the formal, objective, and measurable criteria that can be used to guide selection of appropriate MRs to effectively alleviate the oracle problem. Potential criteria include the code coverage of MRs, the differences in execution profiles (such as branch hits and branch counts) of source and follow-up test cases for an MR [12], the logical hierarchy of MRs, and so on.

A promising direction for future research will be to integrate the selection of "good" MRs with the systematic identification of MRs. The resultant is an advanced technique achieving both MR identification and selection. It would not only identify MRs without the need for existing ones, but could also systematically select a set of MRs that would be most effective in detecting various faults.

4.2 MT as a new test case generation strategy

State of the art. As already discussed, MT was proposed as a test case generation strategy that could be used regardless of whether or not the oracle problem exists. Previous studies have shown that in addition to alleviating the oracle problem, MT is effective at revealing real-life faults, even for widely-used programs, such as the famous Siemens programs [77, 93], C compilers [50, 86], bioinformatics software [16], and wireless embedded systems [49]. Segura et al. [81] reported that MT had detected about 295 real-life faults, emphasizing the effectiveness of the technique. Among these detected faults, two results are worth highlighting for further analysis: the detection of three faults in the Siemens suite and the detection of over 100 faults in two popular C compilers (GCC and LLVM).

MT detected three real-life faults in three out of seven programs [77, 93] in the Siemens suite [43]. The Siemens suite had been extensively used as a benchmark for evaluating many test case selection strategies for the previous two decades. Furthermore, the programs are of relatively small size. It was therefore particularly surprising that such faults had remained undetected for so long in spite of the small program sizes and the thorough testing by a large number of test case selection strategies. This clearly demonstrates that MT complements existing test case selection strategies. The success in revealing these previously undetected faults is due to MT's innovative approach to generating test cases based on a perspective different from those used before. In MT, testers need to consider the necessary properties of the target algorithm — not the implementation. Even without a complete specification, testers can still identify MRs that describe particular properties, and thus can generate test cases that may reveal faults violating these properties. This situation emphasizes the importance of the concept of test case diversity. Programmers may make a variety of errors, including unexpected ones. Correspondingly, test cases should be designed from different perspectives such that they can trigger as many distinct kinds of execution behaviors as possible. Interested readers may consult our work on the role of diversity in effective test cases [21, 22].

Recently, Le et al. [50] developed a technique to test compilers and detected over 100 faults in the popular GCC and LLVM C compilers. The same technique was also applied to detect over 50 new bugs in OpenCL (Open Computing Language) compilers [51]. The technique is basically MT, as observed by different researchers [51, 78], with a specific instance of the following MR: If source programs SP and SP' are equivalent for input I, then their object programs OP and OP', respectively, are also equivalent on I. Their method constructs SP' from SP by removing the statements in SP that are not executed with input I. Compared with the Siemens test suite, the two compilers are extremely large. Although we do not know the testing history of the two compilers, it is very likely that they were tested with fewer testing methods than the Siemens suite — because the latter has been extensively used as a benchmark for evaluating the effectiveness of testing methods. However, since these two compilers are popular, they must have been used extensively, and thus, it is a surprise that so many faults have been detected. This again demonstrates the effectiveness of MT in revealing real-life faults. In fact, Le et al. are not the first researchers to use MT to test compilers. Tao et al. [86] had previously also used MT, but had only found one fault in the GCC compiler and one in the UCC compiler. This dramatic difference in the number of detected faults also emphasizes the significant impact MR choice has on the fault-detection effectiveness of MT.

Compared with other testing strategies, the main MT overheads relate to the identification of MRs, as well as the generation and execution of follow-up test cases. However, a major advantage of MT is that it does not have the scalability problem that has rendered many other testing techniques unusable on large software, as discussed in Section 2.3. Furthermore, the MR that Le et al. [50] used to test the two compilers is remarkably simple, in spite of the technical complexity of compilers, and can be defined without referring to such complex technical content. In fact, the used MR is applicable to compilers for other programming languages, not restricted to the C compilers.

Challenge 3: Effective test case generation. The effectiveness of MT depends on the MRs and MGs used, while follow-up test cases depend on source test cases and the relevant MR. Thus, the effectiveness of MT actually depends on both the MRs and the source test cases. Nevertheless, research has mainly focused on the impact of MRs (as discussed in Section 4.1), with the impact of source test cases on MT's fault-detection effectiveness having somehow been neglected. Previous studies have focused mainly on the identification of "good" MRs, often overlooking the issue of generating "good" test cases — in terms of fault-detection effectiveness. In most previous studies, source test cases were either randomly generated [7, 55], or were special values [19], or both [16]. As observed by Segura et al. [81], 57% of source test cases in previous studies were randomly generated and 34% were from existing test suites. In other words, investigation of the impact of source test cases on MR (and MT) effectiveness is an area yet to be explored. Some initial work in this area has begun, including attempts to generate source test cases using more advanced techniques, such as fault-based testing [28] and ART [5, 7]. The studies so far conducted are still at relatively initial stages, and it is quite challenging to assess and guarantee the effectiveness of test cases generated for MT, which depends on a variety of factors. It will be worthwhile to more deeply investigate how to generate effective source test cases and consequently follow-up test cases that maximize fault detection.

5 EXTENSION OF MT BEYOND TESTING

This section examines how MT has been extended beyond the context of testing, into MR proving (Section 5.1) and as a unified framework for verification, validation, and quality assessment (Section 5.2).

5.1 Proving MRs

State of the art. In MT, MRs are tested and not proven, which means that, even if there is no MR violation, it still cannot be concluded that the program satisfies the relevant MRs for all inputs. A natural line of investigation, therefore, will be to examine how to prove that a program satisfies MRs for the entire input domain — abbreviated as *proving MRs*, hereafter. To the best of our knowledge, very few investigations in this direction have so far been conducted: one by Chen et al. [27, 29] and one by Gotlieb and Botella [37].

Chen et al. [27, 29] developed a *semi-proving* method that uses symbolic analysis techniques to prove MRs. In addition to providing a general framework for proving, they showed that semi-proving can be combined with testing and debugging, as illustrated by the following example.

Example 4. *Consider a program P implementing a function $f(x)$ that has the following MR: $f(k \times x) = k \times f(x)$ (denoted by MR_o), where k is a non-zero integer. Suppose semi-proving has successfully proven that the program P satisfies MR_o on the entire input domain. Now, test P using a concrete test case, such as $x = 2$, and suppose that the output is correct. Then, based on this result and the proven MR, it can be concluded that $P(4), P(6), P(8), \ldots$ must all be correct — even though the program P has never been tested using these concrete test cases.*

As shown above, semi-proving enables extrapolation from the correctness of a program for *tested* inputs to the correctness for related but *untested* inputs, thus combining testing and proving. It was also observed that proving the correctness of a program could sometimes be achieved by proving a set of MRs [27, 29]. In this way, semi-proving provides a new and automated way to do proving. For complex programs where symbolic analysis cannot be applied globally, semi-proving can be performed on a finite set of selected paths, making it a symbolic testing technique. When semi-proving finds that an MR is not satisfied by the program, it provides a constraint on the inputs for which the relevant MR is violated. For example, suppose that the program has two input parameters, a and b, and that the constraint is $a = (2 \times b) + 5$. Whenever the input parameters satisfy the constraint, the MR will be violated. Obviously, such a constraint is more informative than a concrete test case (such as $a = 11$ and $b = 3$) for revealing the nature of the defect.

Gotlieb and Botella [37] used constraint logic programming to generate test cases that cause violations of given MRs. Their testing framework first translates the program under test into an Equivalent Constraint Logic Program over Finite Domains (*eclp*(fd)), and then generates the negation of the given MR, expressed as a goal to solve with the *eclp*(fd). Because a contradiction of the constraint system means that the MR is satisfied, their technique can prove the satisfaction of MRs for certain programs.

Challenge 4: Metamorphic proving. Geller [36] proposed using test cases to prove program correctness by first testing the program using a sample test case, showing that the output is correct, and then proving that the program's output and the specified target function "are perturbed in the same fashion" as the input values change. In this way, one can generalize from the given test case to a larger domain. Although metamorphic relations are obvious candidates for the generalization, how to make use of them for program proving or disproving, in combination with testing, is a challenge. In particular, extensive research is required to balance the fault-detection effectiveness of the MRs (their proving power) with the difficulty level of the proofs.

To elaborate this point, consider again program P in Example 4, which implements the function $f(x)$ that has the MR$_o$: $f(k \times x) = k \times f(x)$, where k is a non-zero integer. We have shown that proving MR$_o$ can be very useful because it enables extrapolation from the program's correctness for a single test case to the program's correctness with infinitely many untested inputs. In practice, however, the verifier might find that MR$_o$ is too difficult to prove for P. In this situation, the verifier should look for other MRs that are easier to prove, such as MR$_n$: $f(-x) = -f(x)$. Although MR$_n$ is weaker than MR$_o$, proving MR$_n$ for P could be more practical, and a successful proof of $P(-x) = -P(x)$ will still be very useful as it will double the effectiveness of concrete test cases.

Many different proving techniques exist, each with advantages and limitations. For the existing MT-based proving techniques [27, 29, 37], their applicability and scalability rely on their related support tools. Further research is needed to identify the usefulness, advantages, and limitations of MR proving techniques beyond symbolic evaluation and constraint logic programming.

5.2 A framework for verification, validation, and quality assessment

State of the art. Software verification checks whether the products of a given development phase (such as design documents or program code) satisfy the specified requirements. Software validation, on the other hand, checks whether these products meet the user's actual needs. Boehm [9] famously explained the difference as questions of "building the product right" (verification) and "building the right product" (validation).

Although MT was originally proposed as a verification technique, it was later also found to be useful for validation. In a study of testing machine learning classifier software [91], it was observed that implementations of two classifiers, k-Nearest Neighbor (kNN) and Naive Bayes, violated some of the identified MRs. Careful investigation later revealed that some of the violated MRs were not actually necessary properties of the target algorithms — but they were properties expected by the users. For example, although users reported expecting that the order of the class labels would not affect the final classification, the kNN algorithm did not have this property, which resulted in MR violations when the implemented program was tested. This observation led to the understanding that MT could also be used as a validation technique — if the MRs are identified based on actual user expectations rather than on the target algorithm.

While verification and validation focus on the functionality and correctness of software, *software quality assessment*, as an activity, covers a much broader range of characteristics than just functional correctness [98]. MT has, for the first time, been formally introduced as a unified framework for software verification, validation, and quality assessment with large scale empirical studies of major search engines (including Google, Bing, and Baidu) [98]. The investigated software quality (sub)characteristics included functional correctness, capacity, operability, user error protection, maturity, effectiveness, and context completeness. As an example, it was found that the search engines under study had performance degradation when searching large domains, which means that MT is useful for assessing software scalability. The main difference is the source of MRs: in verification, they are derived from the specifications; in validation, they are derived from the user expectations; and in quality assessment, they can be defined by various stakeholders.

Consider again the case of search engines [98], which, due to the lack of a tangible test oracle, can be difficult to test or assess. Because knowledge of the algorithms, or detailed system specifications of these search engines (which could be commercial secrets), was not available, a user-oriented approach was adopted to perform MT — MRs were identified from the users' perspective. These MRs reflected what users actually care about and were not based on the algorithms or designs chosen by the search engine developers. On the one hand, they allowed users to validate the search engines and assess their various quality characteristics. On the other hand, the test results were helpful for the developers to reveal defects and weaknesses in the search engines and, hence, to improve the quality of service. The search engine developers could repeat some of the reported MR violations and confirm that they were indeed caused by software faults or design flaws. This means that the user-oriented MRs were also useful for developers conducting verification.

In summary, MRs have evolved from being just the necessary properties of the target algorithm in relation to multiple inputs and their expected outputs (Definition 1), to additionally including the properties expected by users.

Challenge 5: A unified and comprehensive framework. Research into software validation and quality assessment using MT is still at an initial stage, but the ultimate goal should be the development of a comprehensive MT framework supporting verification, validation, and quality assessment. A major task is to formulate MRs not only from the perspective of the target algorithm, but also from various stakeholders' perspectives, including those of developers, user groups, and independent testers. The identification and formalism of MRs can be quite different for various purposes (including verification, validation, and quality assessment), which are associated with requirements in distinct specification paradigms. Hence, it is challenging to develop a unified framework that can capture and express MRs for different purposes and application domains. In particular, it is a very challenging job to propose a specification language that not only supports the unified expression of MRs by different stakeholders for various purposes, but also facilitates the transformation of individual MRs to a set of automated procedures for constructing MT test cases, bearing in mind that the follow-up test cases may depend not only on the source test cases but also their outputs.

Another major task is to involve a variety of quality characteristics and their associated metrics in the framework. Zhou et al. [98] identified five MRs for search engines and showed how they could be used to evaluate some standard quality (sub)characteristics [85], such as functional correctness, operability, and maturity. Although the majority of MT research has focused on the functional correctness of the software under test, it will be necessary to extend further into the broader context of software quality, addressing such aspects as reliability [70], performance [18], and security [20]. The development of MRs to evaluate the different quality characteristics of various software types will be an important job. The characteristics of different software, combined with the multiple aspects of verification and validation activities, will mean that the integration of all these things into a single comprehensive (MT) framework will be both rewarding and challenging.

6 INTEGRATION WITH OTHER TECHNIQUES

State of the art. In addition to alleviating the oracle problem in the context of testing, MT has also been widely applied to address similar problems in other software engineering areas. Because other techniques, such as debugging, analysis, fault tolerance, and program repair, may normally assume the presence of an oracle, integration with MT should extend their applicability, especially when the oracle does not exist. Other than the small constraint of involving at least two inputs, MT is quite straightforward and should easily achieve integration [3, 45, 46, 57, 92, 93]. In fact, the integration process can be facilitated by the following two-component framework:

- The correspondence between a single test case and an MG (which involves multiple test cases); and
- The correspondence between the pass/fail outcome of a test case and the satisfaction/violation of an MR for the relevant MG.

Using this integration framework, the technique under study can be extended through the application of the two mappings with any appropriate modifications to the original technique.

For example, consider the technique of *debugging with slicing* [46, 93], which conventionally works as follows: “If the program is tested with an input that reveals a failure, then we find the relevant slice, called the execution slice, for this failure-causing input, and debug it.” The rationale is that the execution slice must contain the relevant faulty statement. Using the integration framework, we can modify the debugging with slicing technique as follows: “If the program is tested with an MG that reveals the violation of an MR, we find the relevant slice for this *MR-violating MG* and debug it.” A possible way of modification is to replace the execution slice used in the original technique with the union of execution slices for all the test cases (both source and follow-up) in the MG. The rationale is that the faulty statement must be in the union of the MG-related execution slices, thus giving rise to the MR violation. With this modification, the technique can then be extended to application domains without a test oracle.

Consider Spectrum-Based Fault Localization (SBFL) [92] as another example. Given a test suite that contains at least one failure-causing test case, SBFL statistically estimates the likelihood that a program entity (such as a statement) is faulty. SBFL involves examining each statement to determine how many failure-causing and non-failure-causing test cases have executed it as well as how many have not, thereby generating four measures for each statement. The four measures are then used to calculate a risk value, which can be used to prioritize statements for debugging. The reasoning behind SBFL is that (1) a statement executed by more failure-causing test cases is more likely to be faulty and (2) a statement executed by more non-failure-causing test cases is less likely to be faulty.

Using the integration framework, the original SBFL method can be extended in the following three steps. First, “a given test suite with at least one failure-causing test case” becomes “a given set of MGs with at least one MR-violating MG”. Secondly, “a statement executed by a test case” corresponds to “a statement executed by an MG”. Finally, “a statement not executed by any test case” corresponds to “a statement not executed by any test case in any MG”.

The new SBFL process then determines how many MR-violating and non-MR-violating MGs have executed each statement as well as how many have not, thereby generating four new measures for each statement. These four new measures are then used instead of the respective original ones to calculate the risk values, which in turn can be used to prioritize the statements for debugging. The reasoning behind the new technique integrating SBFL and MT is that (1) a statement executed by more MR-violating MGs is more likely to be faulty and (2) a statement executed by more non-MR-violating MGs is less likely to be faulty. In this way, SBFL can be extended to those application domains that face the oracle problem.

Of course, it may not be universally possible to use MT to enhance every relevant method to render it applicable to programs with the oracle problem. Nevertheless, the simplicity of the integration framework makes it generally applicable in the vast majority of cases.

Challenge 6: Development of new concepts. The integration of MT with other software engineering techniques can lead to the development of new concepts, such as metamorphic slicing, which was proposed in recent work on debugging [93]. Slicing is an important concept in program analysis, testing, and debugging. Many slice types have been developed, such as static slices, dynamic slices, execution slices, and conditioned slices [94]. Nevertheless, the slice definitions to date are basically data-oriented or data-driven. Metamorphic slicing has been introduced to integrate MT with debugging and fault localization techniques [92, 93]. A new family of slices has been proposed, including static metamorphic slices, dynamic metamorphic slices, execution metamorphic slices, and conditioned metamorphic slices. Unlike their conventional counterparts, metamorphic slices are not only data-oriented but also property-oriented because they are related to MRs. This has opened a new research area in slicing.

Although many studies integrating MT with other techniques have already been conducted [3, 45, 46, 57, 92, 93], few new concepts have so far been formally developed. Finding a technique to which MT can be applied is the first challenge, after which it may be possible to develop a new concept. Obviously, even when a technique can be integrated with MT, it does not necessarily mean that new concepts will then be developed. Furthermore, the development of new concepts may not be straightforward. Refer to the example of metamorphic slicing. Although Xie et al. [92, 93] only described one execution metamorphic slice construction (through the set union of execution slices of the related MG), there are many possible ways to group the execution slices to form execution metamorphic slices. Generally speaking, the intended application of the metamorphic slices will influence their definition in terms of conventional slices. Clearly, integration of MT with other techniques and the related development of new concepts will be challenging.

Challenge 7: Development of new techniques. Since its first appearance in the literature in 1998, MT has been integrated with many other techniques, resulting in a family of new methods in various areas, including debugging [93], fault localization [3, 92], fault tolerance [57], and program repair [45]. However, some integration attempts face challenging problems.

There are parallels between the use of MT in testing and its use in other software engineering techniques. In the context of testing, for instance, a single test case and its corresponding pass/fail outcome in test result verification relate to an MG and the corresponding MR satisfaction/violation. However, there are some challenging differences when MT is applied in other contexts. A main aim of software testing is to reveal a fault, which, in MT, can be indicated by the violation of an MR. Once an MR is violated, the major task of testing has been fulfilled — it does not matter too much which test cases in the MG are actually related to the fault. In contrast, failure detection is only the starting point in some software engineering areas such as debugging. Precise knowledge of which test cases are failure-causing may be necessary to be able to proceed, such as with debugging [93], fault localization [3, 92], fault tolerance [57], and program repair [45]. This is not a problem for conventional techniques that use single test cases for verification — the pass/fail outcomes simply correspond to the non-failure-causing/failure-causing test cases, respectively. However, with an MR violation, it is only possible to say that at least one test case in the MR-violating MG is related to the fault, unless we do have a test oracle. It is not clear precisely which test case is related. Such a precision problem is an intrinsic characteristic of MT, and is therefore an unavoidable cost when MT is used to address the oracle problem for other software engineering techniques. Consider, for example, fault tolerance techniques. Traditionally, because of the assumption of an oracle's existence, once an input causes an incorrect output, a fault tolerance mechanism is applied to provide an alternative correct output. To address the oracle problem in fault tolerance, one simple strategy of metamorphic fault tolerance [57] works as follows: Multiple inputs are first constructed according to some equality MRs, and then executed simultaneously. Next, the associated outputs are verified against the MRs to decide whether or not the original input (source input in the MT context) results in a "trustworthy" output (in terms of its correctness). If the original output is regarded as untrustworthy, the most trustworthy output is selected from all the outputs associated with the follow-up inputs. A naive mechanism for metamorphic fault tolerance is shown in the following example.

Example 5. *Suppose t_1 is the original input of a system S, for which three equality MRs, namely MR_i, MR_{ii}, and MR_{iii}, have been identified. Suppose further that another three inputs are constructed as follows: t_2 is constructed as the follow-up input based on t_1 as the source input, using MR_i; t_3 is constructed as the follow-up input based on t_2 as the source input, using MR_{ii}; and t_4 is constructed as the follow-up input based on t_1 as the source input, using MR_{iii}. In other words, the MGs for MR_i,*

MR_{ii}, and MR_{iii} are $\langle t_1, t_2 \rangle$, $\langle t_2, t_3 \rangle$, and $\langle t_1, t_4 \rangle$, respectively. (Note that t_1 does not need to be source input for all MRs.)

Consider the following two different scenarios:

- *MR_i and MR_{iii} are satisfied by their corresponding MGs, while MR_{ii} is violated. In such a scenario, since t_1 is not involved in any MR violation, it can be regarded as trustworthy, and its corresponding output (that is, the output of the original input) can be used.*
- *MR_i and MR_{ii} are satisfied by their corresponding MGs, while MR_{iii} is violated. In such a scenario, since t_1 is involved in one MR violation while t_2 is not involved in any MR violation, t_2 can be regarded as more trustworthy than t_1, and its corresponding output should be used.*

However, such a mechanism may result in both false negatives and false positives. On the one hand, a non-failure-causing input involved in an MR-violating MG may be mistakenly judged as untrustworthy and hence discarded — thus a false negative occurs. On the other hand, it is possible to select a failure-causing input as the most trustworthy one and thereby give an incorrect output — thus a false positive occurs. Such an imprecision brings in new challenges, for example, in the accurate evaluation of trustworthiness among multiple inputs and outputs.

In spite of the test case precision challenges, MT has demonstrated its applicability and effectiveness in other software engineering areas [3, 45, 46, 57, 92, 93]. Furthermore, there is great potential to develop new methods to further improve the precision, and thus further enhance the effectiveness. A ranking mechanism, for instance, could be introduced after MT verification. In such a mechanism, individual test cases could be ranked according to their probability of being related to faults (provided that there are statistically sufficient data on the relationships among test cases, MRs, MGs, and the satisfaction/violation outcomes). The ranking results could in turn be used with other software engineering techniques. For example, the test cases most likely related to faults would be the first ones used in the next steps of debugging, fault localization, or program repair. Any resultant methods would no longer be the simple combination of MT and other techniques, but rather new methods, specifically developed and used to be more precise and accurate.

7 MORE RESEARCH OPPORTUNITIES

In addition to the research challenges highlighted in Sections 4 to 6, we next describe seven further opportunities for MT research. This list of opportunities is not exhaustive, but focuses on those research areas we consider most promising. Areas that have already been deeply studied in previous work, such as MT in ubiquitous computing [58, 90], will not be discussed here.

Opportunity 1: Theory of MT. Although extensive studies have been conducted demonstrating the applicability and effectiveness of MT in addressing the oracle problem for software testing and many other software engineering areas, there is a lack of comprehensive work on the fundamental theory of MT. Liu et al. [55], for instance, recommended that a small number of diverse MRs be sufficient by themselves to achieve a fault-detection capability similar to the oracle, and thus to effectively alleviate the oracle problem. However, the concept of diversity was not formally defined, and testers were asked to use their own intuitions to judge the diversity and similarity among MRs. It is therefore not surprising to observe that various testers have different interpretations of diversity, and thus have distinct schemes for classifying MRs — the lack of unified and formulated definitions for diversity has resulted in the ad hoc and arbitrary manner of MR identification and selection.

One possible solution is based on the concepts of category and choice used in METRIC [25], which have been used to create a measure to gauge the dissimilarities among test cases [6]. This metric assesses how different two test cases are based on how many distinct categories and choices they are associated with. In the METRIC framework [25], each MR is associated with a set of categories and choices, so it should be feasible to convert the concept of dissimilarity among test cases into a new metric to assess the diversity among MRs. Such a diversity metric will significantly assist MT research in a number of ways, including helping testers to systematically select a set of diverse MRs that could alleviate the oracle problem effectively [55]. It could also help detect and remove "redundant" MRs — in a group of MRs showing zero diversity with one another, only one such MR would be needed in testing. The diversity metric will also facilitate measurement of the effectiveness of a group of MRs — it is intuitively expected that the more diverse the MRs are, the more effective they will be in alleviating the oracle problem.

In addition to the diversity metric, a lot of work can be done regarding a fundamental theory of MT. Such work will involve investigating the systematic identification of MRs; determining the characteristics of effective MRs; examining how likely a group of MRs mimic a test oracle (if it exists); determining the overall fault-detection effectiveness of MT; exploring the impact of the choice of source test cases on the fault-detection effectiveness of MT; and prioritizing MRs. This theoretical research into MT will enable breakthroughs not only in software testing, but also in the broader area of software engineering, including debugging, proving, specifications engineering, and quality assurance.

Opportunity 2: Teaching and training. As MT has been increasing in popularity, how to teach it to students, professional software engineers and testers, and end users has become an issue of the utmost importance. Teaching experiences by MT researchers [54, 63, 64] indicate that university-level computer science students accept MT and can apply it easily. Reports [87, 88] of how MT, in particular MR identification, has prompted a higher level of student engagement in software testing indicate MT's potential use to encourage student creativity. On the other hand, students have encountered challenges related to the availability of appropriate learning materials and activities. Further work will be required to design the best training materials and methods.

Although various experiences from different universities have shown the ease of teaching and learning MT's basic concepts, which are arguably simple to grasp, a more challenging job will be to improve the learners' ability to derive good and effective MRs, something that will involve a certain degree of art and craftsmanship. Practice and apprenticeship shall play an important role in in-depth teaching and learning of how to effectively conduct MT.

Opportunity 3: New metrics for coverage and confidence. Similar to how the statement coverage criterion enables us to design a set of test cases that execute each reachable statement at least once, an *MR coverage* criterion may guide us to design a set of MGs that verify every MR in question at least once. More specifically, at least one MG should be generated for each MR. MRs are normally identified from specifications, thus, MR coverage can be considered as an additional black-box test adequacy criterion. Furthermore, the MR coverage and white-box coverage criteria are complementary and thus can work together. For example, a set of test cases satisfying the statement coverage can be used as source test cases to construct follow-up test cases based on a set of MRs. Obviously, the resultant set of MGs shall satisfy both the black-box (MR) and white-box (statement) coverage criteria. Unlike statement or branch coverage criteria, however, development of the MR coverage criterion will require that several additional issues be addressed. For example, different people may derive different sets of MRs for the same program — something that is not a problem for the application of statement or branch coverage criteria. The quality and effectiveness of MRs should therefore be considered when applying any MR coverage criterion. One possible way to ensure the quality of MRs used for a coverage criterion is to construct a set of very diverse MRs that achieves a good coverage of the functionalities of the software under test. In other words, the theory of MT in Opportunity 1 may help us improve the effectiveness of the MR coverage criterion.

MRs can also be used as a quality measure for open source software (OSS). Given an OSS project, sets of MRs can be posted for its validation and verification. When examining programs that implement the relevant functionality, users can be guided by information regarding which programs have been verified and validated through which MRs — selecting the programs whose MRs are most relevant, as illustrated in the following example.

Example 6. *Consider an OSS project that implements the sine function. Suppose that two MRs are identified for the function, namely,* MR_a*:* $\sin(-x) = -\sin(x)$ *and* MR_b*:* $\sin(x + 2\pi) = \sin(x)$*. Suppose also that a program S-A in the project has only been tested with* MR_a *(not* MR_b*) and another program S-B has only been tested with* MR_b *(not* MR_a*). If the users are land surveyors, they normally deal with positive (anti-clockwise) and negative (clockwise) angles and are not interested in angles larger than* 2π*. As a result,* MR_a *is more meaningful and program S-A is preferred to S-B. On the other hand, if the users are electrical engineers, they are very likely to use the periodical properties of the sine function. As a result,* MR_b *is more meaningful and program S-B is preferred to S-A.*

Obviously, information about the extent to which an OSS program has been tested is a key guide when choosing which programs to use. From the perspective of program selection, intuitively, users may prefer to know which properties (reflected in the MRs) have been tested and satisfied, rather than how extensively the source code has been executed (as measured, perhaps, by the percentage coverage achieved). For users, satisfying a property may deliver a higher confidence on the software than covering a certain percentage of the code.

These new *MR-based* metrics also provide a new perspective on how to make use of test oracle and any technique addressing the oracle problem. Traditionally, the test oracle and related techniques have only been used for test result verification, but the MR-based metrics may inspire a new research area for measuring the adequacy of a test suite.

Opportunity 4: End-user testing. With the advances in development platforms (such as spreadsheets, MATLAB, and Labview) and human interfaces for advanced systems, end-user programming has been growing at a very fast rate. An increasing number of programs are actually developed by non-IT domain experts rather than professional software engineers. Some of these end-user developed programs are even used in safety-critical systems [48]. However, because end-user programmers do not often have formal software engineering training, it is not reasonable to expect such software to exhibit the same level of quality as that developed by professionals. As a result, end-user software engineering [48] has become a major research area aiming at guaranteeing and improving the quality of end-user developed software.

Software testing is a systematic approach towards software quality, but it is challenging to develop specific testing techniques for end-user programmers. Most testing methods involve a substantial amount of technical software testing knowledge, as well as a general understanding of software engineering. However, because end-user programmers normally have no formal training in software testing or software engineering, it is difficult for them to fully understand the limitations and technical issues of these testing methods. Even if they were able to understand the technical details, it would still be quite challenging for them to implement the methods, which often involve large-scale and highly complex programming, and thus should be done by professional programmers. Furthermore, end-user programmers may not be able to access relevant automated testing tools, even if they are available, because such tools may be quite expensive and not ordinarily affordable. Some of these automated testing tools or methods require quite sophisticated parameter settings in order to ensure cost-effective usage. It may be a very challenging task for end-user programmers to properly set such parameters.

In view of the above problems and constraints, an appropriate testing method should have the following characteristics. First, it must be simple, easily understood, and easy to learn. Secondly, its implementation must be simple. Thirdly, it must be easily automated, or automated tools must be available. Finally, it must be easy for the end users to provide domain-specific information to enhance its effectiveness. As previously explained [23], because MT possesses all four of the characteristics above, it may be the most appropriate testing technique for end-user programmers.

The concept of MT is very simple and can easily be learned in a few hours. The MT testing process can simply be managed by non-professional end-user programmers, who can also prepare test scripts to automate the process. MRs are necessary properties of the target algorithm in relation to multiple inputs and their expected outputs, often identified from the domain knowledge of the system under test. In many cases, therefore, end-users may be even more appropriate or knowledgeable than developers for defining good MRs [55, 98]. In other words, end-user programmers should often be able to effectively use MT without much difficulty. A recent systematic investigation [73] of how to apply MT in end-user testing of spreadsheet systems looked at how a team of non-professionals identified MRs for a set of five spreadsheets with real-life faults. Even though the MRs were identified in an ad hoc way, they were able to detect all the faults, demonstrating the effectiveness of MT as an end-user testing method for such systems. In the future, more research projects should investigate the performance of MT for different development platforms and in various end-user development domains.

Opportunity 5: Cloud and crowd. Orso and Rothermel [69] have advocated the use of cloud computing and crowdsourcing for testing, where it would be natural to embed MT, with the aim of improving the effectiveness and efficiency of MT's implementation. Cloud computing provides new opportunities to enhance the efficiency of testing tasks, including those for MT. In any case, a preliminary project [89] was recently conducted to show the usefulness of cloud-enabled technologies for MT implementation. Much more studies are required to develop a unified cloud-based framework for MT and to investigate its feasibility, applicability, efficiency, and effectiveness. The cloud resources are obtained and allocated through virtual machines (VMs) [10], which are created and destroyed on demand, and only exist for the duration of the testing. When conducting MT in the cloud, different MRs can be used in parallel, thus improving the overall efficiency of MT. Each MR is by itself a standalone entity, so it will be feasible to allocate one VM to each MR for the corresponding test case generation, execution, verification, and test result reporting. It will also be possible to assign a VM specifically for generating source test cases that can then be used by multiple MRs, executing these test cases, and storing their execution results for comparison with those of the follow-up test cases generated in other VMs. The decomposition of a task into sub-tasks for multiple VMs is natural and straightforward in MT. Furthermore, because many cloud-enabled platforms can flexibly allocate computing resources (such as VM locations, time, and types), it is possible to automatically adjust VMs for specific tasks, depending on the resource usage [95]. Since various MRs may require different resources, the flexibility of the cloud-enabled computing platforms will result in optimal resource allocation for MT and ultimately enable a highly efficient MT implementation.

Crowdsourcing is an innovative way of obtaining contributions from many different people, especially through online communities. In MT, the most challenging task is the identification and selection of appropriate MRs, a task that cannot be fully automated, as it requires human intelligence, domain knowledge, and relevant experience. Previous studies [55] have shown that the MRs identified by different individuals naturally contain a degree of diversity, which is strongly correlated with high effectiveness in fault detection. It is thus intuitively appealing to make use of crowdsourcing to brainstorm and decide MRs for a particular system. A variety of personnel can be employed in a crowdsourcing environment, including users, developers, and testers, all of whom can provide various perspectives of domain knowledge for identifying diverse MRs. Since people from different backgrounds need to work together, a major challenge is the need for a formalized framework to support the unified identification and description of MRs. The recent study of MR identification [25] should provide insights in this area.

Opportunity 6: Big data. Big data is popularly defined as data with the 3Vs: high volume, velocity, and variety [70]. It is normally so large and complex that traditional software testing techniques may no longer suffice. Its huge size and various types and formats mean that the oracle problem is prevalent, making testing a major challenge. MT has been recommended as an effective approach for testing big data analytics software [32, 70]. Although Otero and Peter [70] proposed a set of possible MRs related to synonyms, antonyms, and negations, more complicated relations should also be explored, such as those related to subset, intersection, and union. Due to the wide distribution and fast growth of the data, it is difficult to test big data systems at run-time. Setting up sample data is an essential part of the big data testing process. Attempts have been made to construct data samples that reflect the characteristics of the actual data. Alexandrov et al. [1], for example, proposed generation of synthetic data sets based on the actual big data using the data schema, constraints, and other statistical information. When testing big data software, MRs not only cover necessary properties of the system under test, but may also cover properties of the data itself. Similar to the program-related properties, these data-related properties can help produce additional follow-up data to form the sample data, and to verify the test results, especially when the oracle problem exists (which is not rare in big data software). It will be interesting to investigate the extent to which the source and follow-up data, according to various MRs, can together reflect the characteristics of the actual data sets. In addition to the production of sample data, since MRs can relate to the properties of the big data itself, they can help verify, validate, or even prove whether the big data software satisfy properties related to 3Vs, just like what has been done for other software systems[91, 98].

Otero and Peter [70] suggested that MT can be applied beyond testing to other areas of big data software engineering. For example, MRs could be used to create "monitors capable of detecting misbehavior," thus helping assure the reliability of big data software. We believe that MT can be applied to many other aspects of big data. It was recently used to test security software [20], and could therefore naturally be extended into strengthening the protection of big data software from security attacks. Big data analytics involves a variety of learning algorithms, some of which are mathematically complex and not easily understood by programmers or users. The degree to which these algorithms actually meet the users' needs is, therefore, not easily verified. Because MRs are a clear, explicit, and easily understood representation of the necessary properties of the algorithm or user's expectation — with a demonstrated effectiveness in verifying and validating machine learning software [65, 91] — it is natural to expect MT to be applicable in big data software verification and validation.

Opportunity 7: Agile development. Agile development has become one of the most popular paradigms for developing software systems. It normally involves rapid, incremental development, frequent releases of working software, evolutionary requirements improvements, and close collaboration and communication among developers and clients [30]. Although some work has been conducted using MRs in the agile testing of databases [52, 53], the advantages of MT (Section 2.3) suggest that it can easily be applied throughout the entire agile development process.

The most obvious application of MT will be to test software released in every iteration of development, ensuring that each version of the software satisfies the identified MRs. Due to the evolutionary nature of the requirements, the MRs would also require regular updating and fine-tuning. Such updating would not only mean changes to specific MRs, but also the adoption of new MRs and the removal of obsolete ones. Nevertheless, given the close collaboration among stakeholders in agile development, such changes would not represent a difficult task. Furthermore, MRs can provide a simple yet effective way of facilitating the communications between customers and developers — they are the necessary properties that are of the most relevance and interest to the customers, and are clear, non-technical expressions of what must be considered as the software is developed. Moreover, the rapid development and release of successive versions make efficient regression testing critical. If MRs are extensively involved in agile development, then new regression testing techniques involving minimization, prioritization, and augmentation of MRs and MGs will need to be developed and applied. Another potential research direction relates to exploring how to balance the benefits of applying MT in agile development against the cost of maintaining MRs for rapidly changing requirements (especially when an oracle is available). It should be noted that because an MR reflects a specific property, incremental changes in requirements may only cause the updating of a small number of MRs — in other words, the maintenance of MRs in agile development should only incur a small overhead.

8 CONCLUSION

Metamorphic testing (MT) first appeared in 1998 as a methodology for generating follow-up test cases based on successful test cases, guided by some necessary properties of the system under test, called metamorphic relations (MRs). Since then, MT has mainly been used as a simple, but effective, approach to alleviating the oracle problem, with MRs as test result verification mechanisms. MT has successfully detected various faults in a variety of application domains, and advanced techniques have been developed by integrating it with other software engineering methods, often addressing the oracle problem in those other areas. MT has also been applied outside of testing, including in validation, quality assessment, debugging, fault localization, fault tolerance, program repair, and proving.

In this paper, we have reviewed a variety of research topics related to MT, highlighted challenges that need to be addressed, and unveiled some of the most promising opportunities for future MT research. In contrast to — and complementary to — a traditional literature review [81], we have focused on the most important and influential MT studies, providing a more in-depth discussion (including a formal and comprehensive description of MT and a clarification of the major and common misunderstandings of MT) and offering a higher-level vision of MT research and application (including a framework to support integration of MT with other techniques). Our investigation also showed many opportunities to further improve the existing MT research areas, including MR identification, source test case generation, and the application of MT in new domains such as end-user software engineering and big data software. We have also highlighted MT's promise as a novel approach to bolstering other related areas, including measurements for coverage and confidence, cloud-based quality assurance, and agile software development.

MT has evolved from originally defining MRs as necessary properties of the target algorithm in relation to multiple inputs and their expected outputs, to additionally including the properties expected by users. This evolution, of both MT and MRs, is expected to continue.

The concluding statement of this paper is a recommendation to researchers who may be developing new software engineering methods that somehow assume or require a test oracle. It is advisable to consider MT in the development, which may alleviate the oracle requirement, extend the method's scope and applicability, and even facilitate the development of a more comprehensive method.

REFERENCES

[1] Alexander Alexandrov, Christoph Brücke, and Volker Markl. 2013. Issues in big data testing and benchmarking. In *Proceedings of the 6th International Workshop on Testing Database Systems (DBTest '13)*. ACM, New York, NY, 1:1–1:5.

[2] Paul E. Ammann and John C. Knight. 1988. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers* 37, 4, 418–425.

[3] Chittineni Aruna and R. Siva Ram Prasad. 2014. Testing approach for dynamic web applications based on automated test strategies. In *ICT and Critical Infrastructure: Proceedings of the 48th Annual Convention of Computer Society of India, Vol II,* Advances in Intelligent Systems and Computing, Vol. 249. Springer, Berlin, Germany, 399–410.

[4] Earl T. Barr, Mark Harman, Phil McMminn, Muzammil Shahbaz, and Shin Yoo. 2015. The oracle problem in software testing: A survey. *IEEE Transactions on Software Engineering* 41, 5, 507–525.

[5] Arlinta Christy Barus. 2010. *An In-Depth Study of Adaptive Random Testing for Testing Program with Complex Input Types.* Ph.D. Thesis. Faculty of Information and Communication Technologies, Swinburne University of Technology, Melbourne, Australia.

[6] Arlinta Christy Barus, Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, Robert Merkel, and Gregg Rothermel. 2016. A cost-effective random testing method for programs with non-numeric inputs. *IEEE Transactions on Computers* 65, 12, 3509–3523.

[7] Arlinta Christy Barus, Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and Heinz W. Schmidt. 2016. The impact of source test case selection on the effectiveness of metamorphic testing. In *Proceedings of the 1st International Workshop on Metamorphic Testing (MET '16).* ACM, New York, NY, 5–11.

[8] Sami Beydeda. 2006. Self-metamorphic-testing components. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC '06)*, Vol. 1. IEEE Computer Society, Los Alamitos, CA, 265–272.

[9] Barry W. Boehm. 1984. Verifying and validating software requirements and design specifications. *IEEE Software* 1, 1, 75–88.

[10] Rajkumar Buyya, Chee Shin Yeo, Srikumar Venugopal, James Broberg, and Ivona Brandic. 2009. Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility. *Future Generation Computer Systems* 25, 6, 599–616.

[11] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: Three decades later. *Communications of the ACM* 56, 2, 82–90.

[12] Yuxiang Cao, Zhi Quan Zhou, and Tsong Yueh Chen. 2013. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In *Proceedings of the 13th International Conference on Quality Software (QSIC '13).* IEEE Computer Society, Los Alamitos, CA, 153–162.

[13] Wing Kwong Chan, Tsong Yueh Chen, Heng Lu, T. H. Tse, and Stephen S. Yau. 2006. Integration testing of context-sensitive middleware-based applications: A metamorphic approach. *International Journal of Software Engineering and Knowledge Engineering* 16, 5, 677–703.

[14] Wing Kwong Chan, Shing Chi Cheung, and Karl R. P. H. Leung. 2007. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research* 4, 2, 60–80.

[15] Tsong Yueh Chen, Shing Chi Cheung, and Siu Ming Yiu. 1998. *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.

[16] Tsong Yueh Chen, Joshua W. K. Ho, Huai Liu, and Xiaoyuan Xie. 2009. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics* 10, article no. 24.

[17] Tsong Yueh Chen, De Hao Huang, T. H. Tse, and Zhi Quan Zhou. 2004. 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).* Polytechnic University of Madrid, Madrid, Spain, 569–583.

[18] Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and Shengqiong Wang. 2009. Conformance testing of network simulators based on metamorphic testing technique. In *Proceedings of the Joint 11th IFIP WG 6.1 International Conference (FMOODS '09) and 29th IFIP WG 6.1 International Conference on Formal Techniques for Distributed Systems (FORTE '09),* Lecture Notes in Computer Science, Vol. 5522. Springer, Berlin, Germany, 243–248.

[19] Tsong Yueh Chen, Fei-Ching Kuo, Ying Liu, and Antony Tang. 2004. Metamorphic testing and testing with special values. In *Proceedings of the 5th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD '04).* International Association for Computer and Information Science, Mt. Pleasant, MI, 128–134.

[20] Tsong Yueh Chen, Fei-Ching Kuo, Wenjuan Ma, Willy Susilo, Dave Towey, Jeffrey Voas, and Zhi Quan Zhou. 2016. Metamorphic testing for cybersecurity. *Computer* 49, 6, 48–55.

[21] Tsong Yueh Chen, Fei-Ching Kuo, Robert G. Merkel, and T. H. Tse. 2010. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software* 83, 1, 60–66.

[22] Tsong Yueh Chen, Fei-Ching Kuo, Dave Towey, and Zhi Quan Zhou. 2015. A revisit of three studies related to random testing. *Science China Information Sciences* 58, 5, 052104:1–052104:9.

[23] Tsong Yueh Chen, Fei-Ching Kuo, and Zhi Quan Zhou. 2005. An effective testing method for end-user programmers. In *Proceedings of the 1st Workshop on End-User Software Engineering (WEUSE '05).* ACM, New York, NY, 21–25.

[24] Tsong Yueh Chen, Pak-Lok Poon, and T. H. Tse. 2003. A choice relation framework for supporting category-partition test case generation. *IEEE Transactions on Software Engineering* 29, 7, 577–593.

[25] Tsong Yueh Chen, Pak-Lok Poon, and Xiaoyuan Xie. 2016. METRIC: METamorphic Relation Identification based on the Category-choice framework. *Journal of Systems and Software* 116, 177–190.

[26] Tsong Yueh Chen, T. H. Tse, and Yuen Tak Yu. 2001. Proportional sampling strategy: A compendium and some insights. *Journal of Systems and Software* 58, 1, 65–81.

[27] Tsong Yueh Chen, T. H. Tse, and Zhi Quan Zhou. 2002. Semi-proving: An integrated method based on global symbolic evaluation and metamorphic testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA '02)*. ACM, New York, NY, 191–195.

[28] Tsong Yueh Chen, T. H. Tse, and Zhi Quan Zhou. 2003. Fault-based testing without the need of oracles. *Information and Software Technology* 45, 1, 1–9.

[29] Tsong Yueh Chen, T. H. Tse, and Zhi Quan Zhou. 2011. Semi-proving: An integrated method for program proving, testing, and debugging. *IEEE Transactions on Software Engineering* 37, 1, 109–125.

[30] David Cohen, Mikael Lindvall, and Patricia Costa. 2004. An introduction to agile methods. In *Advances in Computers*, Vol. 62. Elsevier, Amsterdam, The Netherlands, 1–66.

[31] Edsger W. Dijkstra. 1972. The humble programmer. *Communications of the ACM* 15, 10, 859–866.

[32] Junhua Ding, Xin-Hua Hu, and Venkat Gudivada. 2017. A machine learning based framework for verification and validation of massive scale image data. *IEEE Transactions on Big Data.* DOI: 10.1109/TBDATA.2017.2680460 .

[33] Guowei Dong, Baowen Xu, Lin Chen, Changhai Nie, and Lulu Wang. 2008. Case studies on testing with compositional metamorphic relations. *Journal of Southeast University (English Edition)* 24, 4, 437–443.

[34] Guowei Dong, Baowen Xu, Lin Chen, Changhai Nie, and Lulu Wang. 2009. Survey of metamorphic testing. *Journal of Frontiers of Computer Science and Technology* 3, 2, 130–143.

[35] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. 2006. Carving differential unit test cases from system test cases. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, New York, NY, 253–264.

[36] Matthew Geller. 1978. Test data as an aid in proving program correctness. *Communications of the ACM* 21, 5, 368–375.

[37] Arnaud Gotlieb and Bernard Botella. 2003. Automated metamorphic testing. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC '03)*. IEEE Computer Society, Los Alamitos, CA, 34–40.

[38] Ralph Guderlei and Johannes Mayer. 2007. Statistical metamorphic testing: Testing programs with random output by means of statistical hypothesis tests and metamorphic testing. In *Proceedings of the 7th International Conference on Quality Software (QSIC '07)*. IEEE Computer Society, Los Alamitos, CA, 404–409.

[39] Richard Hamlet. 2002. Random testing. In *Encyclopedia of Software Engineering*. John Wiley, New York, NY.

[40] Mark Harman, Yue Jia, and Yuanyuan Zhang. 2015. Achievements, open problems and challenges for search based software testing. In *Proceedings of the IEEE 8th International Conference on Software Testing, Verification and Validation (ICST '15)*. IEEE Computer Society, Los Alamitos, CA.

[41] Peifeng Hu, Zhenyu Zhang, Wing Kwong Chan, and T. H. Tse. 2006. An empirical comparison between direct and indirect test result checking approaches. In *Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA '06) in conjunction with the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, New York, NY, 6–13.

[42] Zhan-Wei Hui and Song Huang. 2013. Achievements and challenges of metamorphic testing. In *Proceedings of the 4th World Congress on Software Engineering (WCSE '13)*. IEEE Computer Society, Los Alamitos, CA, 73–77.

[43] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. 1994. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the 16th International Conference on Software Engineering (ICSE '94)*. IEEE Computer Society, Los Alamitos, CA, 191–200.

[44] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, and Zuohua Ding. 2013. Testing central processing unit scheduling algorithms using metamorphic testing. In *Proceedings of the 4th IEEE International Conference on Software Engineering and Service Science (ICSESS '13)*. IEEE Computer Society, Los Alamitos, CA, 530–536.

[45] Mingyue Jiang, Tsong Yueh Chen, Fei-Ching Kuo, Dave Towey, and Zuohua Ding. 2017. A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of Systems and Software* 126, 127–140.

[46] Hao Jin, Yanyan Jiang, Na Liu, Chang Xu, Xiaoxing Ma, and Jian Lu. 2015. Concolic metamorphic debugging. In *Proceedings of the IEEE 39th Annual International Computers, Software and Applications Conference (COMPSAC '15)*, Vol. 2. IEEE Computer Society, Los Alamitos, CA, 232–241.

[47] Upulee Kanewala, James M. Bieman, and Asa Ben-Hur. 2016. Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels. *Software Testing, Verification and Reliability* 26, 3, 245–269.

[48] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad Myers, Mary Beth Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. 2011. The state of the art in end-user software engineering. *ACM Computing Surveys* 43, 3, 21:1–21:44.

[49] Fei-Ching Kuo, Tsong Yueh Chen, and Wing K Tam. 2011. Testing embedded software by metamorphic testing: A wireless metering system case study. In *Proceedings of the IEEE 36th Conference on Local Computer Networks (LCN '11)*. IEEE Computer Society, Los Alamitos, CA, 291–294.

[50] Vu Le, Mehrdad Afshari, and Zhendong Su. 2014. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '14)*. ACM, New York, NY, 216–226.

[51] Christopher Lidbury, Andrei Lascu, Nathan Chong, and Alastair F. Donaldson. 2015. Many-core compiler fuzzing. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '15)*. ACM, New York, NY, 65–76.

[52] Mikael Lindvall, Dharmalingam Ganesan, Ragnar Árdal, and Robert E. Wiegand. 2015. Metamorphic model-based testing applied on NASA DAT: An experience report. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, Vol. 2. IEEE, Piscataway, NJ, 129–138.

[53] Mikael Lindvall, Dharmalingam Ganesan, Sigurthor Bjorgvinsson, Kristjan Jonsson, Haukur Steinn Logason, Frederik Dietrich, and Robert E. Wiegand. 2016. Agile metamorphic model-based testing. In *Proceedings of the 1st International Workshop on Metamorphic Testing (MET '16)*. ACM, New York, NY, 26–32.

[54] Huai Liu, Fei-Ching Kuo, and Tsong Yueh Chen. 2010. Teaching an end-user testing methodology. In *Proceedings of the 23rd IEEE Conference on Software Engineering Education and Training (CSEE&T '10)*. IEEE Computer Society, Los Alamitos, CA, 81–88.

[55] Huai Liu, Fei-Ching Kuo, Dave Towey, and Tsong Yueh Chen. 2014. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Transactions on Software Engineering* 40, 1, 4–22.

[56] Huai Liu, Xuan Liu, and Tsong Yueh Chen. 2012. A new method for constructing metamorphic relations. In *Proceedings of the 12th International Conference on Quality Software (QSIC '12)*. IEEE Computer Society, Los Alamitos, CA, 59–68.

[57] Huai Liu, Iman I. Yusuf, Heinz W. Schmidt, and Tsong Yueh Chen. 2014. Metamorphic fault tolerance: An automated and systematic methodology for fault tolerance in the absence of test oracle. In *Companion Proceedings of the 36th International Conference on Software Engineering (ICSE Companion '14)*. ACM, New York, NY, 420–423.

[58] Heng Lu, Wing Kwong Chan, and T. H. Tse. 2006. Testing context-aware middleware-centric programs: A data flow approach and an RFID-based experimentation. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering (SIGSOFT '06/FSE-14)*. ACM, New York, NY, 242–252.

[59] Xiaoli Lu, Yunwei Dong, and Chao Luo. 2010. Testing of component-based software: A metamorphic testing methodology. In *Proceedings of the 7th International Conference on Ubiquitous Intelligence and Computing and the 7th International Conference on Autonomic and Trusted Computing (UIC/ATC '10)*. IEEE Computer Society, Los Alamitos, CA, 272–276.

[60] Ieng Kei Mak. 1997. *On the Effectiveness of Random Testing*. Master's Thesis. Department of Computer Science, The University of Melbourne, Melbourne, Australia.

[61] L. I. Manolache and D. G. Kourie. 2001. Software testing using model programs. *Software: Practice and Experience* 31, 13, 1211–1236.

[62] Johannes Mayer and Ralph Guderlei. 2006. An empirical study on the selection of good metamorphic relations. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC '06)*, Vol. 1. IEEE Computer Society, Los Alamitos, CA, 475–484.

[63] Kunal Swaroop Mishra and Gail E. Kaiser. 2012. *Effectiveness of teaching metamorphic testing*. Technical Report CUCS-020-12. Department of Computer Science, Columbia University, New York, NY.

[64] Kunal Swaroop Mishra, Gail E. Kaiser, and Swapneel Kalpesh Sheth. 2013. *Effectiveness of teaching metamorphic testing, Part II*. Technical Report CUCS-022-13. Department of Computer Science, Columbia University, New York, NY.

[65] Christian Murphy, Gail Kaiser, Lifeng Hu, and Leon Wu. 2008. Properties of machine learning applications for use in metamorphic testing. In *Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE '08)*. Knowledge Systems Institute Graduate School, Skokie, IL, 867–872.

[66] Christian Murphy, M. S. Raunak, Andrew King, Sanjian Chen, Christopher Imbriano, Gail Kaiser, Insup Lee, Oleg Sokolsky, Lori Clarke, and Leon Osterweil. 2011. On effective testing of health care simulation software. In *Proceedings of the 3rd Workshop on Software Engineering in Health Care (SEHC '11)*. ACM, New York, NY, 40–47.

[67] Shin Nakajima and Hai Ngoc Bui. 2016. Dataset coverage for testing machine learning computer programs. In *Proceedings of the 2016 Asia-Pacific Software Engineering Conference (APSEC '16)*. IEEE Computer Society, Los Alamitos, CA, 297–304.

[68] Rafael A. P. Oliveira, Upulee Kanewala, and Paulo A. Nardi. 2015. Automated test oracles: State of the art, taxonomies, and trends. In *Advances in Computers*, Vol. 95. Elsevier, Amsterdam, The Netherlands, 113–199.

[69] Alessandro Orso and Gregg Rothermel. 2014. Software testing: A research travelogue (2000–2014). In *Proceedings of the Future of Software Engineering (FOSE '14)*. ACM, New York, NY, 117–132.

[70] Carlos E. Otero and Adrian Peter. 2015. Research directions for engineering big data analytics software. *IEEE Intelligent Systems* 30, 1, 13–19.

[71] Krishna Patel and Robert M. Hierons. submitted for publication. A systematic literature review on testing and debugging non-testable systems. Available at http://people.brunel.ac.uk/~csstrmh/Intt/synth.pdf.

[72] Mauro Pezzè and Cheng Zhang. 2014. Automated test oracles: A survey. In *Advances in Computers*, Vol. 95. Academic Press, Waltham, MA, 1–48.

[73] Pak-Lok Poon, Fei-Ching Kuo, Huai Liu, and Tsong Yueh Chen. 2014. How can non-technical end users effectively test their spreadsheets? *Information Technology and People* 27, 4, 440–462.

[74] Pak-Lok Poon, Huai Liu, and Tsong Yueh Chen. 2017. Error trapping and metamorphic testing for spreadsheet failure detection. *Journal of Organizational and End User Computing* 29, 2, 25–42.

[75] Laura L. Pullum and Ozgur Ozmen. 2012. Early results from metamorphic testing of epidemiological models. In *Proceedings of the 2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom '12)*. IEEE Computer Society, Los Alamitos, CA, 62–67.

[76] Arvind Ramanathan, Chad A. Steed, and Laura L. Pullum. 2012. Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics. In *Proceedings of the 2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom '12)*. IEEE Computer Society, Los Alamitos, CA, 68–73.

[77] Peifeng Rao, Zheng Zheng, Tsong Yueh Chen, Nan Wang, and Kai-Yuan Cai. 2013. Impacts of test suite's class imbalance on spectrum-based fault localization techniques. In *Proceedings of the 13th International Conference on Quality Software (QSIC '13)*. IEEE Computer Society, Los Alamitos, CA, 260–267.

[78] John Regehr. 2014. Finding compiler bugs by removing dead code. http://blog.regehr.org/archives/1161.

[79] David S. Rosenblum. 1995. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering* 21, 1, 19–31.

[80] Md. Shaik Sadi, Fei-Ching Kuo, Joshua W. K. Ho, Michael A. Charleston, and Tsong Yueh Chen. 2011. Verification of phylogenetic inference programs using metamorphic testing. *Journal of Bioinformatics and Computational Biology* 9, 6, 729–747.

[81] Sergio Segura, Gordon Fraser, Ana B. Sánchez, and Antonio Ruiz-Cortés. 2016. A survey on metamorphic testing. *IEEE Transactions on Software Engineering* 42, 9, 805–824.

[82] Sergio Segura, Robert M. Hierons, David Benavides, and Antonio Ruiz-Cortés. 2010. Automated test data generation on the analyses of feature models: A metamorphic testing approach. In *Proceedings of the 3rd International Conference on Software Testing, Verification and Validation (ICST '10)*. IEEE Computer Society, Los Alamitos, CA, 35–44.

[83] Sergio Segura, Robert M. Hierons, David Benavides, and Antonio Ruiz-Cortés. 2011. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology* 53, 3, 245–258.

[84] Chang-Ai Sun, Guan Wang, Baohong Mu, Huai Liu, Zhaoshun Wang, and Tsong Yueh Chen. 2011. Metamorphic testing for web services: Framework and a case study. In *Proceedings of the 2011 IEEE International Conference on Web Services (ICWS '11)*. IEEE Computer Society, Los Alamitos, CA, 283–290.

[85] Systems and software engineering: Systems and software Quality Requirements and Evaluation (SQuaRE): System and software quality models. ISO/IEC 25010:2011, ISO.

[86] Qiuming Tao, Wei Wu, Chen Zhao, and Wuwei Shen. 2010. An automatic testing approach for compiler based on metamorphic testing technique. In *Proceedings of the 2010 Asia Pacific Software Engineering Conference (APSEC '10)*. IEEE Computer Society, Los Alamitos, CA, 270–279.

[87] Dave Towey and Tsong Yueh Chen. 2015. Teaching software testing skills: Metamorphic testing as vehicle for creativity and effectiveness in software testing. In *Proceedings of the 2015 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE '15)*. IEEE Computer Society, Los Alamitos, CA, 161–162.

[88] Dave Towey, Tsong Yueh Chen, Fei-Ching Kuo, Huai Liu, and Zhi Quan Zhou. 2016. Metamorphic testing: A new student engagement approach for a new software testing paradigm. In *Proceedings of the 2016 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE '16)*. IEEE Computer Society, Los Alamitos, CA, 228–235.

[89] Michael Troup, Andrian Yang, Amir Hossein Kamali, Eleni Giannoulatou, Tsong Yueh Chen, and Joshua W. K. Ho. 2016. A cloud-based framework for applying metamorphic testing to a bioinformatics pipeline. In *Proceedings of the 1st International Workshop on Metamorphic Testing (MET '16)*. ACM, New York, NY, 33–36.

[90] T. H. Tse, Stephen S. Yau, Wing Kwong Chan, Heng Lu, and Tsong Yueh Chen. 2004. Testing context-sensitive middleware-based software applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC '04)*, Vol. 1. IEEE Computer Society, Los Alamitos, CA, 458–465.

[91] Xiaoyuan Xie, Joshua W. K. Ho, Christian Murphy, Gail E. Kaiser, Baowen Xu, and Tsong Yueh Chen. 2011. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software* 84, 4, 544–558.

[92] Xiaoyuan Xie, W. Eric Wong, Tsong Yueh Chen, and Baowen Xu. 2011. Spectrum-based fault localization: Testing oracles are no longer mandatory. In *Proceedings of the 11th International Conference on Quality Software (QSIC '11)*. IEEE Computer Society, Los Alamitos, CA, 1–10.

[93] Xiaoyuan Xie, W. Eric Wong, Tsong Yueh Chen, and Baowen Xu. 2013. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology* 55, 5, 866–879.

[94] Baowen Xu, Ju Qian, Xiaofang Zhang, Zhongqiang Wu, and Lin Chen. 2005. A brief survey of program slicing. *ACM SIGSOFT Software Engineering Notes* 30, 2, 1–36.

[95] Iman I. Yusuf, Ian E. Thomas, Maria Spichkova, Steve Androulakis, Grischa R. Meyer, Daniel W. Drumm, George Opletal, Salvy P. Russo, Ashley M. Buckle, and Heinz W. Schmidt. 2015. Chiminey: Reliable computing and data management platform in the cloud. In *Proceedings of the 37th International Conference on Software Engineering (ICSE '15)*, Vol. 2. IEEE, Piscataway, NJ, 677–680.

[96] Jie Zhang, Junjie Chen, Dan Hao, Yingfei Xiong, Bing Xie, Lu Zhang, and Hong Mei. 2014. Search-based inference of polynomial metamorphic relations. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, NY, 701–712.

[97] Zhenyu Zhang, Wing Kwong Chan, T. H. Tse, and Peifeng Hu. 2009. Experimental study to compare the use of metamorphic testing and assertion checking. *Journal of Software* 20, 10, 2637–2654.

[98] Zhi Quan Zhou, Shaowen Xiang, and Tsong Yueh Chen. 2016. Metamorphic testing for software quality assessment: A study of search engines. *IEEE Transactions on Software Engineering* 42, 3, 264–284.

[99] Zhi Quan Zhou, ShuJia Zhang, Markus Hagenbuchner, T. H. Tse, Fei-Ching Kuo, and Tsong Yueh Chen. 2012. Automated functional testing of online search services. *Software Testing, Verification and Reliability* 22, 4, 221–243.

[100] Hong Zhu. 2015. JFuzz: A tool for automated Java unit testing based on data mutation and metamorphic testing methods. In *Proceedings of the 2nd International Conference on Trustworthy Systems and Their Applications (TSA '15)*. IEEE Computer Society, Los Alamitos, CA, 8–15.

[101] Hong Zhu, Patrick A. V. Hall, and John H. R. May. 1997. Software unit test coverage and adequacy. *ACM Computing Surveys* 29, 4, 366–427.

Received May 2017; revised August 2017; accepted September 2017