How Effectively does Metamorphic Testing Alleviate the Oracle Problem?

Huai Liu, *Member, IEEE*, Fei-Ching Kuo, *Member, IEEE*, Dave Towey, *Member, IEEE*, and Tsong Yueh Chen, *Member, IEEE*

***Abstract*— In software testing, something which can verify the correctness of test case execution results is called an oracle. The oracle problem occurs when either an oracle does not exist, or exists but is too expensive to be used. Metamorphic testing is a testing approach which uses metamorphic relations, properties of the software under test represented in the form of relations among inputs and outputs of multiple executions, to help verify the correctness of a program. This paper presents new empirical evidence to support this approach, which has been used to alleviate the oracle problem in various applications and to enhance several software analysis and testing techniques. It has been observed that identification of a sufficient number of appropriate metamorphic relations for testing, even by inexperienced testers, was possible with a very small amount of training. Furthermore, the cost-effectiveness of the approach could be enhanced through the use of more diverse metamorphic relations. The empirical studies presented in this paper clearly show that a small number of diverse metamorphic relations, even those identified in an ad hoc manner, had a similar fault-detection capability to a test oracle, and could thus effectively help alleviate the oracle problem.**

***Index Terms*— Software testing, test oracle, oracle problem, metamorphic testing, metamorphic relation.**

I. Introduction

The scale and use of software systems around the world have been growing exponentially, yet at the same time, reports of problems due to software faults have also been growing. Software quality assurance has become one of the most important areas in the software industry as well as in the academic community. Software testing, a major approach in software quality assurance, is widely acknowledged as a critical activity and a main research focus in software engineering [18]. One of the objectives of software testing is to detect as many software faults as possible, and to do so as quickly as possible [33].

Although many effective test case selection strategies have been proposed [11], [14], the majority of these strategies rely on the availability of a test oracle [20], a mechanism which can systematically verify the correctness of a test result for any given test case (input). Only with a test oracle can it be clearly claimed that the output of the program under test passes or fails for any given input. In many situations, however, a test oracle either does not exist, or is too expensive to be used. This problem, referred to as the *oracle problem*, is a fundamental challenge for software testing, because it significantly restricts the applicability and effectiveness of most test case selection strategies.

Metamorphic testing [5] is an approach to alleviating the oracle problem. In metamorphic testing, some necessary properties (hereafter referred to as *metamorphic relations*) are identified for the software under test, usually from the software specifications. These metamorphic relations provide a new perspective on verifying test results. Traditionally, after a test case is executed, its corresponding test output is verified using a test oracle. Unlike traditional testing, metamorphic testing always involves multiple test case executions, with their corresponding outputs being verified using the metamorphic relations rather than a test oracle.

Metamorphic testing has been applied in various application domains, successfully detecting faults [6], [25], [32], [37], [40], and has also been integrated with other software analysis and testing technologies to extend their applicability to those programs without test oracles [4], [10], [41]. The effectiveness of metamorphic relations has been studied, with attempts made to establish guidelines for the selection of "good" metamorphic relations [7], [29]. Studies [21], [42] have also been conducted comparing metamorphic testing with other techniques, such as assertion checking, for alleviating the oracle problem.

There remain, however, some as yet unanswered, fundamental research questions related to metamorphic testing, such as: whether, and to what extent, metamorphic relations can alleviate the oracle problem; how many metamorphic relations are required to match the fault-detection effectiveness of a test oracle; and what are the key factors that influence the effectiveness of metamorphic testing. This paper presents an investigation of these questions through a series of empirical studies. The rest of the paper is organized as follows: In Section II, the procedure and background information for metamorphic testing are presented. Work related to metamorphic testing is discussed in Section III. In Section IV, the three fundamental research questions in this study are explained. Details of the empirical studies are presented in Section V, the results of which, and the answers to the research questions, are reported in Section VI. In Section VII, potential threats to the validity of this study are discussed. Finally, Section VIII summarizes the paper.

II. Metamorphic Testing

The basic steps for implementing metamorphic testing are as follows:

1) Some necessary properties of the software under test are identified (normally extracted from the specifications), and

This research project is supported by an Australian Research Council Discovery Grant.

H. Liu is with Australia-India Centre for Automation Software Engineering, RMIT University, Melbourne 3001 VIC, Australia (e-mail: huai.liu@rmit.edu.au).

F.-C. Kuo and T. Y. Chen are with the Faculty of Information and Communication Technologies, Swinburne University of Technology, Hawthorn 3122 VIC, Australia (e-mail: dkuo@swin.edu.au; tychen@swin.edu.au).

D. Towey is with the Division of Computer Science, The University of Nottingham Ningbo China, Ningbo 315100 Zhejiang, China (e-mail: dave.towey@nottingham.edu.cn).

Digital Object Indentifier 10.1109/TSE.2013.46

represented in the form of relations, referred to as metamorphic relations. Each metamorphic relation involves multiple test case inputs and their corresponding outputs.
2) Some test cases, referred to as the source test cases, are generated using traditional test case selection strategies.
3) New test cases, called the follow-up test cases, are constructed from the source test cases according to the metamorphic relations.
4) Both source and follow-up test cases are applied to the software under test.
5) The test case outputs are checked against the relevant metamorphic relations to confirm whether the relations are satisfied, or have been violated.

The following example illustrates how metamorphic testing works. Suppose that a program P searches for the shortest path between two nodes in an undirected graph. One metamorphic relation of P is that if the start and goal nodes are swapped, the length of the shortest path should remain unchanged. Suppose that a source test case (G, a, b) is selected according to some testing strategy, where G is an undirected graph, and a and b are the start and goal nodes, respectively. According to the metamorphic relation, a follow-up test case (G, b, a) can be constructed. After the execution of both test cases, the outputs can be checked against the relation by confirming whether or not $|P(G, a, b)| = |P(G, b, a)|$ is satisfied (where $|\cdot|$ denotes the length of a path). If the relation has been violated, it can be concluded that P is faulty.

Metamorphic testing is not only simple in concept, but once the metamorphic relations have been identified, it can also be easily automated. Since metamorphic testing first appeared, it has been widely applied in various application domains. Murphy et al. [32] developed a framework for implementing metamorphic testing in machine learning; their framework includes a degree of automation, and can also be applied in other disciplines. Segura et al. [37] applied metamorphic testing to the analysis of feature models. Furthermore, metamorphic testing has detected real-life faults in a number of programs, including a bioinformatics program [6], two C compilers [40], a wireless metering system [25], and three programs in the popular Siemens suite (`schedule`, `schedule2`, and `print_tokens`) [35], [41] which have been extensively used and tested in the literature [13]. The detection of these faults proves that metamorphic testing has brought a new perspective to testing, not only for test result verification, but also for test case generation. In other words, metamorphic testing can be viewed as a test case selection method complementary to existing methods.

Metamorphic testing has also been integrated with other software analysis and testing techniques. Beydeda [4], for example, presented an approach integrating the self-testing COTS components method with metamorphic testing, which, because metamorphic testing provided an automatic way for test result verification, significantly enhanced the COTS self-testability. Chen et al. [10] integrated metamorphic testing with symbolic execution, resulting in a method which can help verify program correctness with respect to certain necessary properties. The method also provided some important and useful information to support debugging. Gotlieb and Botella [17] developed an automatic testing framework by combining metamorphic testing with constraint logic programming. Xie et al. [41] applied metamorphic testing to fault localization, and proposed a methodology that supports spectrum-based fault localization without the need for a test oracle. Their investigations included the development of metamorphic slicing, an integration of metamorphic relations with slicing techniques.

III. Related work

An approach to addressing the oracle problem may involve the construction of oracles from formal specifications. Hierons [20], for example, developed algorithms for two types of conformance relations to test physically distributed systems using the framework of finite state machines. However, such an approach requires specifications expressed in a formal notation, which is not always feasible. Compared with this, metamorphic testing is a more generic approach, because it is applicable regardless of how the specifications are written.

The assertion checking technique [36] uses assertions, normally embedded in the source code during the programming phase. These assertions are constraints on specific portions of the source code, which, when the program is executed, help detect software faults that violate the constraints at runtime. With assertion checking, a fault could be revealed with a single execution of the program under test, whereas metamorphic testing always requires multiple executions. The fault-detection capability of assertion checking has been compared with that of metamorphic testing [21], [42]. It has been demonstrated that metamorphic testing consistently detects more faults than assertion checking, but may incur additional overheads. Different from these studies, in the current paper we attempt to directly compare the fault-detection effectiveness of metamorphic testing with that of a test oracle, and thus answer more fundamental research questions, such as, whether or not metamorphic testing really provides an effective mechanism for imitating a test oracle. As will be discussed in Section V-C, random testing with a base program as the oracle is used to simulate the automatic test result verification against an oracle, and thus provides a benchmark for evaluating the effectiveness of metamorphic testing.

Another approach to alleviating the oracle problem has been to use multiple versions of the program under test to play the role of test oracle. Manolache and Kourie [28] achieved this using N-version programming [2], which was originally designed as a fault tolerance technique. They developed a so-called M-mp testing strategy, in which M (M $\geq$ 1) "model programs" are used together with the program under test to construct "an approximate test oracle." Such an approach, however, is not always reliable: Knight and Leveson [24] have reported that different versions of a program may not be developed independently. Furthermore, there is also the possibility of faults common to all versions appearing. Ammann and Knight [1] proposed the fault tolerance technique of data diversity for situations where one and only one version of a program exists. Their method requires an input t to be "reexpressed" into t', where t' and t contain the same information, but in different forms. For example, since a property of sine is that for a given angle t, $\sin(t) = \sin(\pi - t)$; if P is a program which calculates the sine value of t, and outputs a value greater than 1 (which obviously shows that P does not compute this input correctly), then input t' is set to $(\pi - t)$, and executed, hopefully resulting in a correct output. However, this technique was designed with the assumed existence of a test oracle, and, given the nature of fault tolerance, has the constraint that only the equality relation is allowed.

Recently, the mutation analysis technique [12] was used to help construct test oracles. Staats et al. [38] proposed a mutation-based method for automatically selecting some variables to construct a test oracle, however, such a method assumed the existence of an oracle — among all well-defined internal state or output variables, it selects some which show high effectiveness in killing mutants. Fraser and Zeller [16] investigated how to generate unit tests and oracles based on a mutation technique, but the oracles defined in their method are effectively a list of assertions. Our study focuses on whether, and to what extent, metamorphic testing effectively alleviates the oracle problem. Thus, rather than comparing metamorphic testing with other techniques, we concentrate on evaluating its effectiveness against the benchmark provided by a simulated and automated oracle — which is actually random testing applied to a base program, as explained in Section V-C.

Some relatively simple ways to detect faults without a complete test oracle also exist. Testers can use as inputs, some "special" values for which the expected outputs are well-known (e.g., $\sin(\pi/2)$ must be 1). However, the applicability of such special case testing is very limited. Another method is to reveal faults by causing the program under test to crash (have an execution error, such as segmentation fault, infinite loop, divide-by-zero, etc). Although such an approach has successfully revealed faults [15], [30], [31], only certain types of faults can be detected.

In addition to research into the application of metamorphic testing, studies of its core components, metamorphic relations, have also been conducted. Chen et al. [7] investigated how to distinguish metamorphic relations with better fault-detection potential. They suggested that testers should understand not only the application domain, but also the algorithm's structure, and reported that for "good" metamorphic relations, the execution behavior of the source test case should be very different from that of the follow-up test case. In addition, Mayer and Guderlei [29] examined several determinant computation programs to identify their metamorphic relations, finding that relations with rich semantic properties had better fault-detection effectiveness.

Although there have already been many studies of various aspects of metamorphic testing, to date, no work has been done to evaluate how effectively metamorphic relations may be able to approach the fault-finding efficiency of a test oracle. This paper attempts to answer the research questions surrounding this issue through a series of empirical studies.

IV. Research questions

This section summarizes the fundamental research questions for metamorphic testing, and how they will be answered through the empirical studies.

- RQ1: How effectively can metamorphic relations detect software faults?
  Metamorphic relations, the core components of metamorphic testing, provide a test result verification mechanism which can imitate a test oracle. Their fault-detection effectiveness is the major factor determining to what extent metamorphic testing can alleviate the oracle problem. In the empirical studies presented in this paper, the fault-detection effectiveness of metamorphic relations was evaluated from the following perspectives: the performance of each individual metamorphic relation was measured according to the number of faults it revealed; the total number of faults revealed by all identified metamorphic relations for a subject program was calculated; and finally, the relationship between the number of metamorphic relations used and the overall fault-detection effectiveness was analyzed. Through the investigation of RQ1, not only can we qualitatively assess how effectively metamorphic testing alleviates the oracle problem, but we can also quantitatively evaluate how many metamorphic relations would normally be required to effectively imitate a test oracle, based on which some guidelines could be provided for applying metamorphic testing in practice. As detailed in Section V-C, the effectiveness of metamorphic relations is evaluated against two benchmarks: random testing with and without an oracle. If a set of metamorphic relations can deliver a fault-detection effectiveness much higher than that of random testing without oracle, and similar to that of random testing with an oracle, they could be considered to effectively imitate a test oracle.
- RQ2: How capable are testers of alleviating the oracle problem using metamorphic testing?
  Obviously, the applicability and effectiveness of a testing method depend on human factors, such as how easily testers can learn the method, and how effectively they can apply it. Previous studies [21], [42] have shown that it is not difficult for testers to understand the basic concept of metamorphic testing, and be able to identify some appropriate metamorphic relations. In this study, an in-depth analysis was conducted of how easily metamorphic testing is understood and applied, and also of the extent to which testers could use metamorphic testing to alleviate the oracle problem. This was done as follows: the number of faults detected by the metamorphic relations identified by each individual tester was measured; the overall performance of individual testing teams, each consisting of several testers, was examined; and the relationship between the fault-detection effectiveness and the number of testers involved was investigated.
- RQ3: How can the cost-effectiveness of metamorphic testing be optimized?
  The two previous research questions focus on fault-detection effectiveness, which is measured as the number of detected faults. However, high fault-detection effectiveness may not be useful if its cost is too high. A good testing method should have high cost-effectiveness, that is, it should detect as many faults as possible at a relatively low cost. The cost-effectiveness of metamorphic testing depends on the number of metamorphic relations and their fault-detection effectiveness. This study investigated the fundamental factors affecting the cost-effectiveness of metamorphic relations; a better understanding of these factors should make it possible to optimize the cost-effectiveness of metamorphic testing.

V. Experiment

An empirical analysis was adopted to answer the research questions in Section IV. The design and settings of the experiments are described in this section.

A. Subject programs and mutant generation

When we selected the subject programs, a consideration was that for appropriate identification of proper metamorphic relations, the testers may need a significant amount of specific domain

knowledge. Because there are many application domains (such as scientific computing, financial calculations, web services, database applications, image processing, clinical systems, etc), it is practically infeasible, if not impossible, to conduct an investigation for general domains — it is extremely difficult to recruit a sufficient number of testers with in-depth domain knowledge for the various applications — therefore, our investigation needed to be constrained to just a few domains. In the experiments, all the testers were university students in computer science and/or software engineering without any commercial experience. We selected the algorithmic programs for which the testers had sufficient domain knowledge to identify metamorphic relations. In addition, the subject programs were selected such that they were neither too complex nor too simple: If too complex, the experiment would have required a prohibitively long time, and this study aimed to complete both the training and experimental application, for each individual participant, within a single day. If too simple, the faults could easily have been detected by any testing method, and might thereby undermine the validity of the experiment.

Five Java programs representing different application areas were selected as the subjects of the experiments (see Table I). Understanding their specifications only required some basic knowledge of data structures and search algorithms, which the university students in these studies had already acquired. Mutants for the subject programs were generated automatically using `muJava` [27]. Since the focus was on the basic functionality of each program, not the class interfaces, only the "traditional" mutation operators (such as arithmetic operator replacement, relational operator replacement, etc) were used to generate the mutants, each of which contained a single fault. It should also be noted that, like other specification-based techniques, the effectiveness of metamorphic testing is hindered if mistakes exist in the software's specifications. Nevertheless, the subject programs' specifications were well-defined, with little ambiguity, and were examined very carefully before being distributed to the testers. Therefore, the possibility of problems in the specifications, and their potential impact on our study, were minimized.

As will be discussed in Sections V-B and VI-A, the metamorphic relations identified in this study detected real-life faults in the original `MultipleKnapsack` [26] and `SparseMatrixMultiply` [22] programs. These faulty programs were fixed, and the corrected versions were then used as subjects in the study.

B. Metamorphic relation identification

For each subject program, two testing teams were assigned to identify its metamorphic relations. Each team was composed of four to seven members, who were postgraduate or senior undergraduate students in computer science and/or software engineering from the same university.

Although some of the students participating in the study had already learned some basic software testing concepts, they had neither the knowledge of metamorphic testing nor the practical experience in testing. Before working on the subject programs, all students were given a three-hour training session covering basic metamorphic testing concepts and some examples of metamorphic relations for applications other than the subject programs. The training session consisted of a ninety-minute tutorial and a ninety-minute exercise. In the tutorial, a trainer (a co-author of this paper) first gave a one-hour presentation on metamorphic testing and metamorphic relations, during which several examples were used to illustrate the metamorphic relation identification. The trainer then hosted a thirty-minute discussion session, during which the students were encouraged to raise any questions and to discuss the related topics. Next, to examine their understanding of the training content, the students were given an exercise involving the calculation of average, standard deviation, and median values for a set of real numbers. They were then asked to work individually to identify as many metamorphic relations as possible within a period of one hour. In the final thirty minutes, the trainer commented on the identified relations, discussing and providing feedback to the students.

After the three-hour training, each student was next assigned up to two subject programs, for each of which they were asked to identify as many metamorphic relations as possible within ninety minutes. Students worked individually and independently during the metamorphic relation identification process: They were not permitted to communicate with each other. At this point, students who had been given the same subject program, and who came from the same university, were considered to form a testing team (even though they had not collaborated on the relations identification). To minimize the learning effects, we allocated the subject programs according to the following criteria: (1) Subject programs were given in different orders — for example, one team was required to first work on the `MultipleKnapsack` program, and then, after this, on a different program; at the same time, another team was given a different program, after they finished with which, they then worked on `MultipleKnapsack`. (2) Each pair of testing teams had at most one common subject program — no two testing teams were allocated exactly the same subject programs. For example, a testing team from University A worked on the `SparseMatrixMultiply` and `FindKNN` programs; while another testing team, from University B, worked on `FindKNN` and `SetCover`.

After the identification process, the trainer (different testing teams may be associated with different trainers) checked all identified metamorphic relations, keeping the valid relations, and discarding all others. The checked results were then also confirmed by other trainers/co-authors, to further assure the correctness. Table II summarizes the results of the metamorphic relation identification process. In the table, the totals for metamorphic and invalid relations refer to distinct relations — for `MinimizeDFA`, for example, the testing team from University B identified eleven distinct metamorphic relations and two distinct invalid relations, and the team from University C identified ten distinct metamorphic relations and two distinct invalid relations, but in total, only sixteen distinct metamorphic relations and three distinct invalid relations were identified. From the table it can be observed that, on average, each student was able to identify two to six distinct metamorphic relations for each subject program, which is consistent with other studies involving ad hoc identification of metamorphic relations [21], [42]. It can also be observed that each testing team could identify seven to eighteen distinct metamorphic relations for each program. Table II also shows that for each subject program, only up to three of the relations identified by the testers were not valid metamorphic relations. The average number of invalid relations was always less than one per tester, per subject program, and usually no greater than 0.5. Moreover, it was also easy for the trainer (who is

TABLE I
SUBJECT PROGRAM INFORMATION

Program	Line of code	Number of mutants	Basic functionality
`FindKNN`	153	698	Finding the k nearest neighbors of a sample point [34]
`MinimizeDFA`	929	1,660	Minimizing a deterministic finite automaton [23]
`MultipleKnapsack`	808	1,905	Solving the multiple knapsack problem [26]
`SparseMatrixMultiply`	259	212	Multiplying two sparse matrices [22]
`SetCover`	211	258	Solving the set coverage problem using a greedy algorithm [3]

very familiar with the specifications) to identify invalid relations, usually in less than a minute, simply by reading them. In other words, the impact of invalid relations was very small. As a reminder, the identification process was conducted in an ad hoc manner: The students were not taught any systematic method to generate the metamorphic relations.

All identified metamorphic relations were first verified against the original programs. Surprisingly, this verification process revealed two real-life faults in `MultipleKnapsack`, and one fault in `SparseMatrixMultiply` (details are reported in Section VI-A) — in other words, the metamorphic relations, even defined in such an ad hoc way, were very effective at revealing real faults! After fixing these faults, since the corrected versions of these two programs, and the other three original subject programs, satisfied all metamorphic relations, they were referred to as the base programs in the experiments.

C. Random testing with and without a test oracle

The fault-detection effectiveness of metamorphic testing was evaluated against two benchmarks: random testing without a test oracle, referred to as RT in this paper; and random testing with a test oracle, referred to as RTo in this paper. Both of these methods show a high degree of testing automation — test case generation and test result verification are automated — with little human bias, and thus provide simple, but fair, comparison benchmarks for evaluating how effectively metamorphic testing alleviates the oracle problem.

Intuitively speaking, the fault-detection effectiveness of RT should be the lower bound for the effectiveness of metamorphic testing: If the performance of metamorphic testing is similar to that of RT, then metamorphic testing cannot effectively alleviate the oracle problem. In the absence of a test oracle, as was the case with RT, a fault could only be revealed when the program crashed (refer to Section III for the definition of "crash"). Such a "crash only" verification scheme has been widely used in research into random testing without a test oracle [15], [30], [31]. Since, in addition to crashing the program, metamorphic testing uses the violation of metamorphic relations to detect faults, intuitively speaking, it should not perform worse than RT.

On the other hand, RTo provides an upper bound for the fault-detection effectiveness of metamorphic testing: If a number of metamorphic relations can collectively detect a similar number of faults to RTo, then they could be considered to effectively imitate a test oracle. In RTo, the base program served as a test oracle to verify the results of mutants, as has been commonly adopted in other experiments using mutation analysis techniques [12]. In addition to crashing, for certain test cases, some mutants produced different outputs to the base program, in which case these mutants were said to be "killed" by these test cases. Both crashing and killing implied the detection of faults.

In the testing, the following categories of fault-detection were of interest: (i) crashed only mutants that were not killed by any other test cases (hereafter referred to as *crashed mutants*); (ii) crashed mutants that were also killed by some other test cases (hereafter referred to as *crashed and killed mutants*); and (iii) non-crashed mutants that were killed by some test cases (hereafter referred to as *killed mutants*). RT could only identify mutants of category (i) (that is, crashed mutants), while both RTo and metamorphic testing could identify mutants of all three categories (that is, killed or crashed mutants). Note that RTo and metamorphic testing had slightly different meanings of "kill": RTo killed a mutant when the mutant had an output different from that of the corresponding base program which was used as the test oracle; metamorphic testing killed a mutant when a metamorphic relation was violated by the outputs of some related source and follow-up test cases for that mutant.

Suppose that n_c mutants were crashed by RT, n_{kc} mutants were killed or crashed by RTo, and n_{MT} mutants were killed or crashed by metamorphic testing using a set of metamorphic relations. In order to quantitatively evaluate how effectively metamorphic testing imitates a test oracle, we introduce an *oracle imitation measure* (M_{oim}), defined as follows.

$$M_{oim} = \frac{n_{MT} - n_c}{n_{kc} - n_c}. \quad (1)$$

Note that Equation 1 is applicable when $n_{kc} > n_c$. Theoretically speaking, although extremely unlikely, it is possible for n_{kc} to equal n_c, which would imply that all faults can be revealed by crashing alone, and thus neither an automated oracle nor metamorphic relations would improve the fault-detection effectiveness. If M_{oim} approaches 0 (that is, $n_{MT} \approx n_c$), it means that metamorphic testing could not effectively alleviate the oracle problem. On the other hand, if M_{oim} approaches 1 (that is, $n_{MT} \approx n_{kc}$), it implies that metamorphic testing could effectively imitate a test oracle. However, it should be pointed out that, as explained in Section VI, because metamorphic testing and RTo use different test cases in our experiments, metamorphic testing was able to detect more faults than RTo (that is, M_{oim} was greater than 1 in some cases); in other words, the fault-detection effectiveness of RTo is only considered to be the theoretical upper bound.

D. Test case generation

For each subject program, one thousand test cases were generated randomly according to uniform distribution using the pseudorandom number generator provided in the Java standard library. By uniform distribution, we mean that all possible program inputs had the same probability of being selected as test cases. For instance, the `MultipleKnapsack` program accepts three sets of integers as input: two n-tuple sets $P = \{p_1, p_2, \cdots, p_n\}$

TABLE II
IDENTIFICATION OF METAMORPHIC RELATIONS

Program	Team	Number of testers in the team	Number of metamorphic relations identified		Number of invalid relations identified	
			Team Total	Average per tester	Team Total	Average per tester
FindKNN	University A	5	10	3.8	0	0
	University B	5	11	4.2	2	0.4
	total	10	16	4	2	0.2
MinimizeDFA	University B	6	11	2.8	2	0.33
	University C	7	10	2.3	2	0.29
	total	13	16	2.5	3	0.31
MultipleKnapsack	University D	5	17	6	1	0.2
	University A	4	18	5.3	2	0.5
	total	9	27	5.7	3	0.33
SparseMatrixMultiply	University D	5	18	4	0	0
	University A	5	7	2.8	0	0
	total	10	22	3.4	0	0
SetCover	University A	4	11	3.8	1	0.25
	University B	5	15	3.8	3	0.6
	total	9	18	3.8	3	0.44

and $W = \{w_1, w_2, \cdots, w_n\}$, which respectively represent the profits and weights of n items to be selected, and one m-tuple set $C = \{c_1, c_2, \cdots, c_m\}$, which represents the capacities of m knapsacks to hold the selected items. The following procedure was implemented to generate the random test cases for MultipleKnapsack: First, randomly select the values for n and m. Second, randomly select the values of p_i and w_i for each of the n items ($i = 1, 2, \cdots, n$). Third, randomly select the value of c_j for each of the m knapsacks ($j = 1, 2, \cdots, m$). An example of the test cases randomly generated for MultipleKnapsack is as follows: $P = \{95, 30, 93, 72, 19, 14, 68, 31, 56, 99\}$, $W = \{46, 70, 91, 17, 80, 39, 88, 35, 24, 62\}$, and $C = \{113, 129, 150\}$, where $n = 10$ and $m = 3$. All one thousand random test cases were used in the execution of RT and RTo.

Metamorphic testing involves two types of test cases, the source and the follow-up. In the experiments, some of the one thousand random test cases were selected as the source test cases, based on which follow-up test cases were generated. In order to have a fair comparison with RT and RTo, both of which used one thousand random test cases, the total number of source and follow-up test cases was kept as close as possible to one thousand, without exceeding it, and the number of common test cases for metamorphic testing and RT/RTo was maximized.

We applied the following settings in the experiments: Suppose a metamorphic relation requires m_s source test cases and m_f follow-up test cases. When only one metamorphic relation was used in testing, the first $\left\lfloor \frac{1000}{1+m_f/m_s} \right\rfloor$ random test cases were used as source test cases. For example, suppose that MRa was used, and it required two source test cases and one follow-up test case. First, $\left\lfloor \frac{1000}{1+1/2} \right\rfloor = 666$ random test cases were used as source test cases. Then, 333 follow-up test cases were generated, each of which was constructed from two source test cases according to MRa. Thus, a total of 999 (666 source + 333 follow-up) test cases were used for testing with MRa.

When the testing of a subject program used multiple metamorphic relations, MR1, MR2, $\cdots$, and MRn, each of which required m_s^i source test cases and m_f^i follow-up test cases ($i = 1, 2, \cdots, n$), the first $\left\lfloor \frac{1000}{1+\sum_{i=1}^{n} m_f^i/m_s^i} \right\rfloor$ random test cases were used as source test cases. Based on the source test cases, follow-up test cases were constructed according to each of the relevant metamorphic relations. For example, suppose that two metamorphic relations, MRb and MRc, were used: MRb required one source test case and one follow-up test case; and MRc required one source test case and two follow-up test cases. First, $\left\lfloor \frac{1000}{1+(1+2)} \right\rfloor = 250$ random test cases were selected as source test cases. Next, each source test case was then used to construct one follow-up test case according to MRb, and two follow-up test cases according to MRc. This resulted in a total of $1,000$ test cases ($250 \times$ (1 source + 1 follow-up for MRb + 2 follow-up for MRc)).

As shown, this test case generation arrangement ensured that up to a thousand test cases were generated and executed for each run of metamorphic testing on an individual subject program.

VI. EXPERIMENTAL RESULTS

A. Detection of real-life faults

The metamorphic relations identified by the testers were first tested on the original programs, surprisingly revealing real-life faults in the MultipleKnapsack and SparseMatrixMultiply programs, as detailed in this section.

In the Java code of SparseMatrixMultiply [22], the 194$^{\text{th}}$ line was "ic[0] = 1;", but should have been "ic[1] = 1;". This fault was revealed by 17 of the 22 identified metamorphic relations. Nine of the ten testers who had been assigned the SparseMatrixMultiply program identified metamorphic relations capable of revealing this fault.

There were two faults found in the Java code of MultipleKnapsack [26]: one was on the 95$^{\text{th}}$ line, which was "q += origw[j]" but should have been "q += origp[j]"; and the other was on the 190$^{\text{th}}$ line, which was "idex1 = aux[i]" but should have been "idx1 = aux[1]". Of all 27 metamorphic relations identified for MultipleKnapsack, 11 could reveal the first fault, and 19 could reveal the second. All nine testers who had been assigned MultipleKnapsack identified metamorphic relations that revealed the second fault; and seven testers identified relations that revealed the first.

Previous studies of metamorphic testing have also reported revealing real-life faults, even for well-tested software, including three programs in the popular Siemens suite (`schedule`, `schedule2`, and `print_tokens`) [35], [41], C compilers [40], bioinformatics programs [6], and wireless embedded systems [25]. The detection of these real-life faults implies that in addition to alleviating the oracle problem, metamorphic testing is also an effective test case selection method complementary to existing methods.

As a reminder, the mutation analysis in the following Sections (VI-B to VI-D) was based on the corrected versions of `MultipleKnapsack` and `SparseMatrixMultiply`, and the original versions of the other three programs. In other words, the base programs satisfied all the metamorphic relations before mutation began, and hence were used as test oracles in this analysis.

B. RQ1: Fault-detection effectiveness of metamorphic relations

1) Fault-detection effectiveness of individual metamorphic relations: Fig. 1 summarizes the fault-detection effectiveness of individual metamorphic relations compared with RT; with RTo; and with all identified metamorphic relations. In Fig. 1 (also in Fig. 3 and Fig. 4), crashed mutants are displayed as white boxes; crashed and killed mutants are displayed as black boxes on top of white ones; and killed mutants are displayed as grey boxes on top of black ones. As a reminder, since RT referred to random testing without a test oracle, it could only identify crashed mutants, while both RTo and metamorphic testing could kill mutants in addition to crashing.

Because different test cases were used, the numbers of crashed mutants for metamorphic testing were not necessarily equal to those for RT (although they were quite similar): RT used the whole pool of the one thousand randomly generated test cases; while metamorphic testing used part of this random pool as source test cases, and generated new follow-up test cases. As can be observed from Fig. 1, apart from two metamorphic relations (MR13 for `SparseMatrixMultiply` and MR10 for `FindKNN`), the overwhelming majority of the relations (97 out of 99) both killed and crashed mutants. In other words, it was very likely that metamorphic testing using a single metamorphic relation had a higher fault-detection effectiveness than RT.

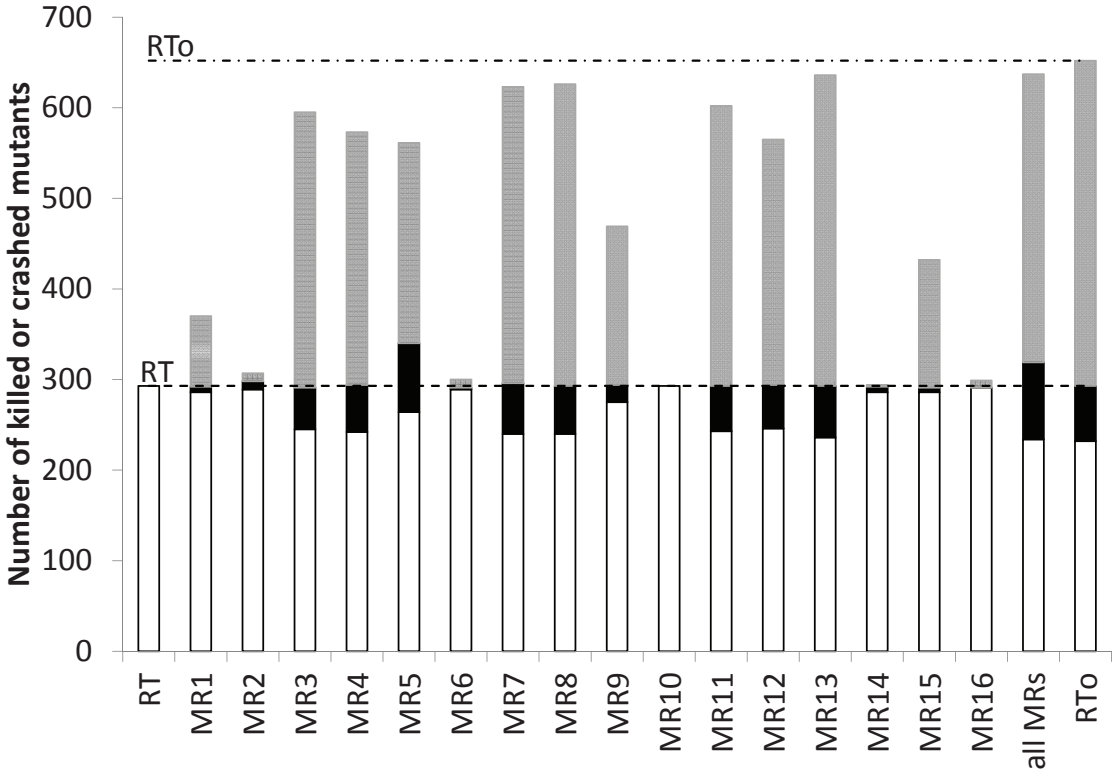
Hypothesis testing was conducted to verify whether this observation was statistically significant. The significance level was set to 0.05, and the null hypotheses (H_0) were that each individual metamorphic relation had similar fault-detection effectiveness to RT. When the p-value was smaller than the significance level of 0.05, the null hypothesis was rejected; otherwise, it was accepted. The two-tailed t-tests results are reported in the 2^{nd} column of Table III, in each cell of which, the decision (reject or accept) was given based on the p-value represented by the number in parentheses. It was shown that for each subject program, the null hypothesis was rejected. Based on these t-test results, and the data shown in Fig. 1, it can be concluded that even though the metamorphic relations were identified in an ad hoc way, each one by itself had significantly higher fault-detection effectiveness than RT. In summary, in this instance of the oracle problem, metamorphic relations definitely helped to reveal more faults than RT, which depended entirely on program "crashing" to reveal faults.

The t-tests for the comparison between individual metamorphic relations and RTo are summarized in the 3^{rd} column of Table III. Obviously, an arbitrary metamorphic relation could not be expected to outperform a test oracle, therefore an investigation was conducted into how well individual metamorphic relations could alleviate the oracle problem. A histogram analysis was used to quantitatively compare the fault-detection capabilities of the individual metamorphic relations and a test oracle. For each subject program, 10 groups of metamorphic relations were defined as follows: Each of the first 9 groups was defined as the group of metamorphic relations that individually has the value of $M_{oim} \in [(i-1) \times 0.1, i \times 0.1)$, where $i = 1, 2, \cdots, 9$. The 10^{th} group contains the metamorphic relations that have $M_{oim} \geq 0.9$. Technically speaking, each group (with the exception of Group 10) represents a 10% difference in effectiveness between RT and RTo. The grouping of metamorphic relations is summarized in Table IV, where the number of metamorphic relations in each group for each program is given. For example, the value of "5" in the entry corresponding to Group 7 and the program `MinimizeDFA` means that there were five metamorphic relations for `MinimizeDFA`, each of which outperformed RT by 60% to 70% of the difference in effectiveness between RT and RTo.

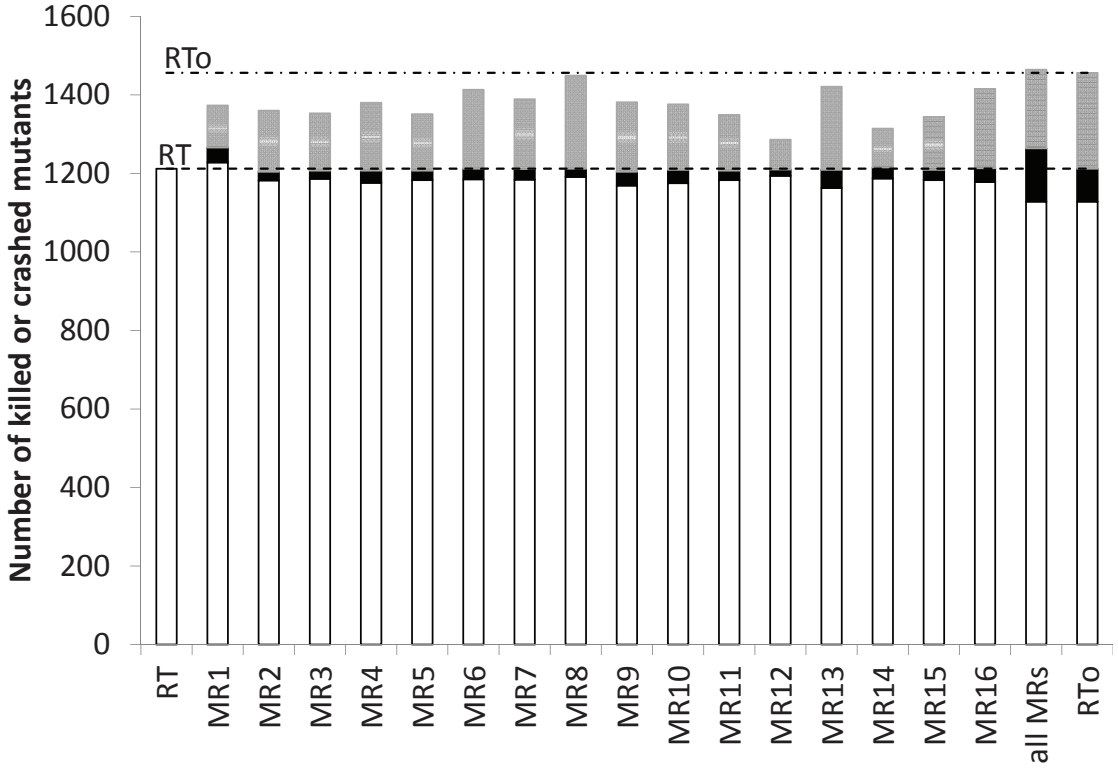
Based on Table IV, it can be observed that there were 19 metamorphic relations (Group 10, 19.19% of all), each of which outperformed RT by at least 90% of the difference in effectiveness between RT and RTo. Since 55.56% $((6+10+8+12+19)/99)$ of the metamorphic relations are in Groups 6-10, this means that over half of identified metamorphic relations each achieved a fault-detection effectiveness at least half way between that of RT and that of RTo.

2) Fault-detection effectiveness when using all identified metamorphic relations: We compared the fault-detection effectiveness of using all identified metamorphic relations with that of RTo and of RT, as summarized in Table V. It can be observed that the use of all identified metamorphic relations always had much higher fault-detection effectiveness than RT. It can also be observed that when using all identified metamorphic relations, metamorphic testing killed or crashed a similar (or sometimes even larger) number of mutants compared with RTo. For one program (`FindKNN`), the fault-detection effectiveness when using all identified metamorphic relations was only marginally lower than that of RTo ($0.95 < M_{oim} < 1$). For another program (`SetCover`), the use of all identified metamorphic relations and RTo had exactly the same fault-detection effectiveness ($M_{oim} = 1$). For the remaining three programs (`MinimizeDFA`, `MultipleKnapsack`, and `SparseMatrixMultiply`), the use of all identified metamorphic relations was able to outperform RTo. Note that although RTo had been expected to play the role of an upper bound on performance, it was possible for metamorphic testing in this experiment to detect more faults than RTo because they used different test cases. In summary, the collection of all identified metamorphic relations, even though they were identified in an ad hoc way, can be regarded as an effective imitation for a test oracle.

3) Relationship between fault-detection effectiveness and the number of metamorphic relations used in metamorphic testing: Fig. 2 reports how many mutants for individual subject programs were killed or crashed when a certain number of metamorphic relations were used together in the testing. In Fig. 2, each boxplot represents the statistical distribution of the number of mutants



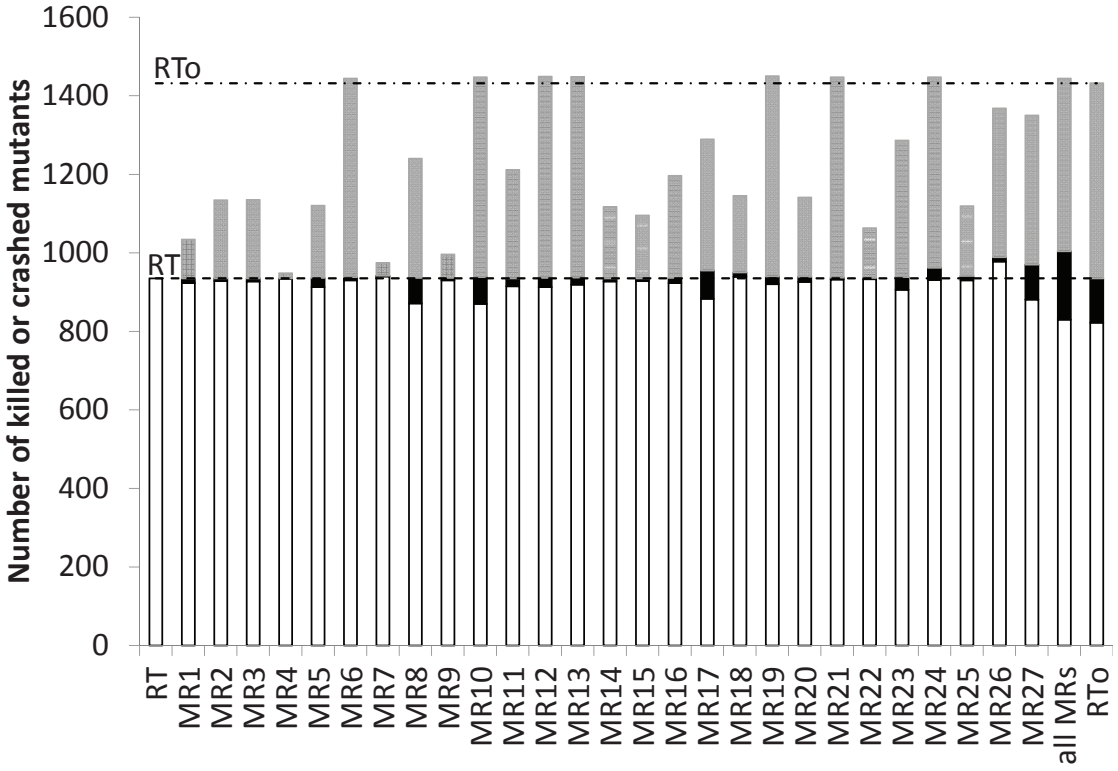
(a) `FindKNN`



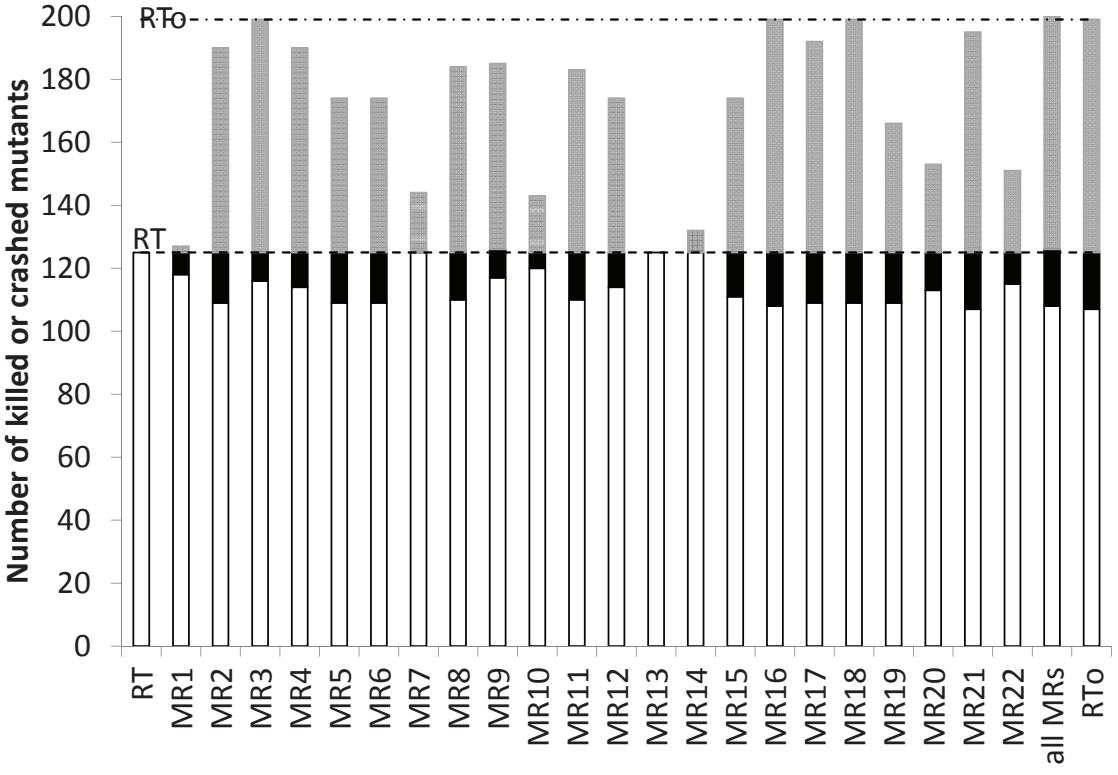
(b) `MinimizeDFA`

killed or crashed by a given number of metamorphic relations (note that for this given number, there were various possible groups of metamorphic relations). The upper and lower bounds of the box denote the third and first quartile of the number of killed or crashed mutants, respectively, while the middle line inside the box represents the median value. The top and bottom

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication.



(c) `MultipleKnapsack`



(d) `SparseMatrixMultiply`

whiskers denote the maximum and minimum values, respectively, and a square dot denotes the mean number of mutants killed or crashed by a given number of metamorphic relations. For ease of comparison, the fault-detection effectiveness of RTo and RT are given as the horizontal dot lines in Fig. 2.

Fig. 2 shows a trend that when more metamorphic relations

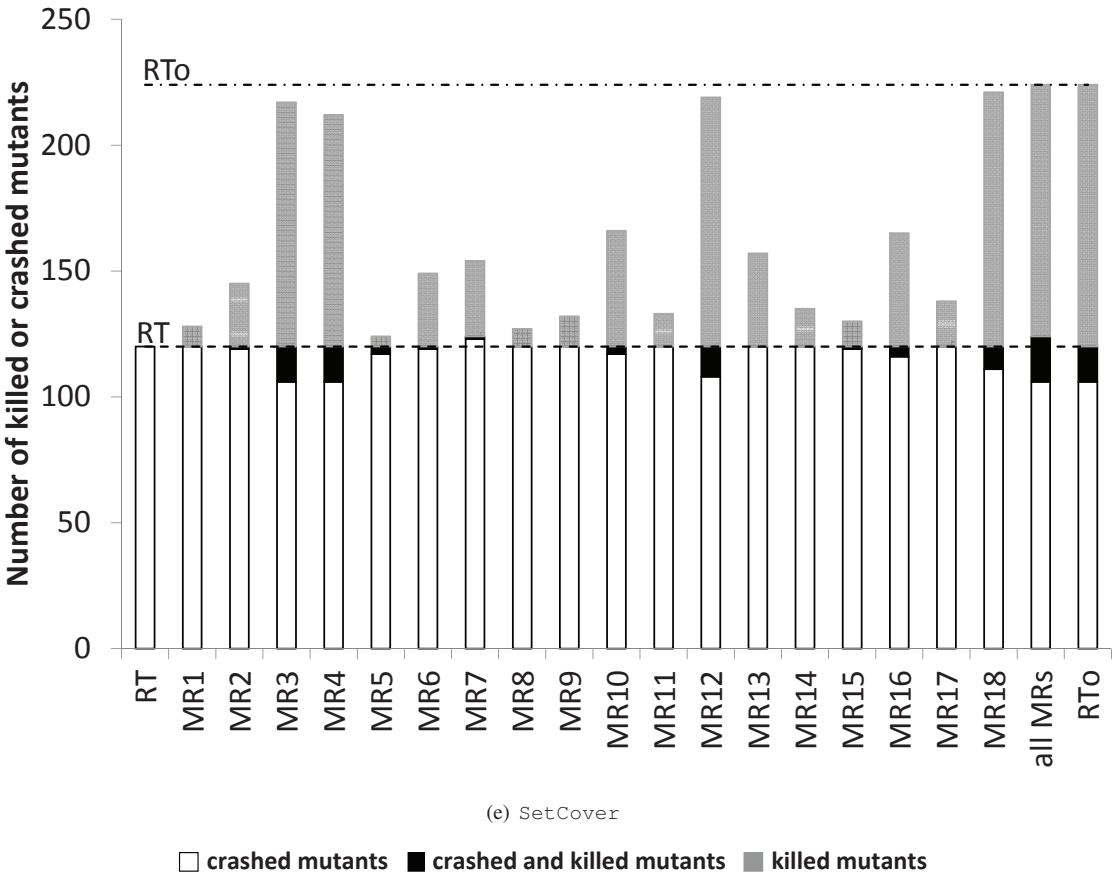


Fig. 1. Relationship between individual metamorphic relations and the number of killed/crashed mutants

TABLE III
T-TESTS FOR COMPARING INDIVIDUAL METAMORPHIC RELATIONS WITH RT AND RTO

Program	Decision on the comparison with	
	RT	RTo
FindKNN	REJECT (1.33×10^{-4})	REJECT (1.19×10^{-4})
MinimizeDFA	REJECT (1.13×10^{-10})	REJECT (6.70×10^{-7})
MultipleKnapsack	REJECT (1.69×10^{-9})	REJECT (8.59×10^{-7})
SparseMatrixMultiply	REJECT (1.51×10^{-8})	REJECT (3.83×10^{-5})
SetCover	REJECT (2.07×10^{-4})	REJECT (3.59×10^{-7})

H_0: Each individual metamorphic relation had similar fault-detection effectiveness to RT/RTo.

were used, not only were more mutants killed or crashed, but also the variation between the fault-detection capabilities of various groups of the same number of metamorphic relations decreased. In other words, with an increase in the number of metamorphic relations, the fault-detection effectiveness was not only increased, but also stabilized.

Based on the data in Fig. 2, it is possible to calculate the average number (n_{MR}) of metamorphic relations required to outperform RT by at least 90% of the difference in effectiveness between RT and RTo. It was found that for FindKNN, MinimizeDFA, MultipleKnapsack, SparseMatrixMultiply, and SetCover, n_{MR} was 5 (31.25% of 16), 4 (25% of 16), 4 (14.81% of 27), 3 (13.63% of 22), and 6 (33.33% of 18), respectively. These results imply that even though 16 to 27 metamorphic relations were identified for each subject program in this investigation, it may not have been cost-effective to use all of them in the testing. For each program under investigation, there was a very good chance of outperforming RT by at least 90% of the difference in effectiveness between RT and RTo by using just three to six of the metamorphic relations. In other words, to imitate a test oracle, it may have been more cost-effective for metamorphic testing to use an arbitrary choice of at most a third of all the identified metamorphic relations — even though these relations were identified by different testers without

TABLE IV
GROUPING OF METAMORPHIC RELATIONS, COMPARING WITH RT AND RTO

Group	FindKNN	Minimize DFA	Multiple Knapsack	SparseMatrix Multiply	Set Cover	Total number of metamorphic relations in a group
1	5	0	2	3	4	14
2	0	0	2	0	4	6
3	1	0	1	2	2	6
4	1	1	4	2	2	10
5	1	1	4	0	2	8
6	0	4	2	0	0	6
7	0	5	1	4	0	10
8	3	1	2	2	0	8
9	2	3	2	4	1	12
10	3	1	7	5	3	19
Total number of metamorphic relations for a program	16	16	27	22	18	99

TABLE V
FAULT-DETECTION EFFECTIVENESS FOR ALL METAMORPHIC RELATIONS COMPARED WITH RTO/RT

Program	M_{oim} when using all metamorphic relations
FindKNN	0.9582
MinimizeDFA	1.0328
MultipleKnapsack	1.0241
SparseMatrixMultiply	1.1757
SetCover	1

extensive experience in testing.

C. RQ2: Capabilities of testers

1) Capability of individual testers: The fault-detection effectiveness of metamorphic relations identified by individual testers for each subject program is reported in Fig. 3, which, for ease of comparison, also includes the results for using all metamorphic relations, for RTo, and for RT.

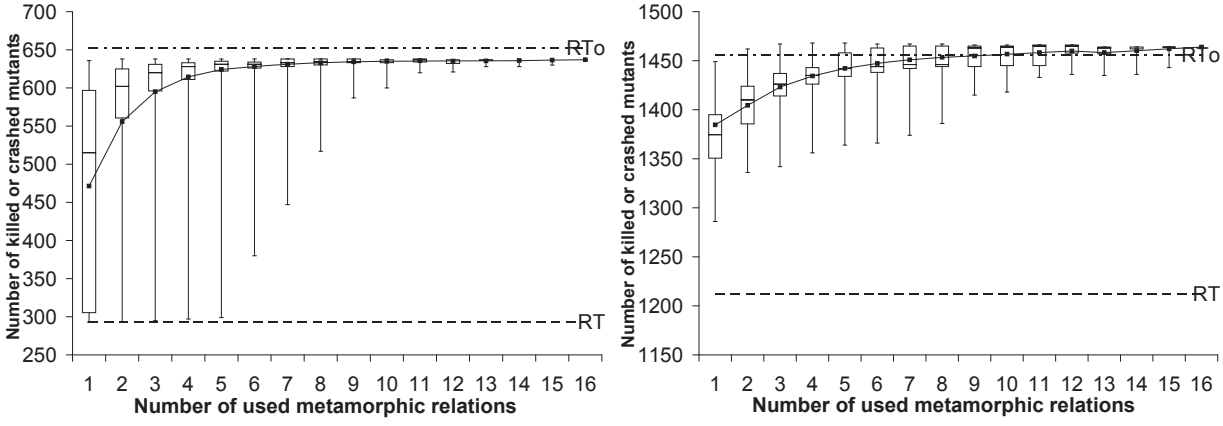
Fig. 3 clearly shows that each tester was able to identify metamorphic relations that were sufficient by themselves to reveal more faults than RT. It can also be observed that the fault-detection effectiveness of the identified metamorphic relations varied from tester to tester. Table VI reports the results of two-tailed t-tests conducted to compare the fault-detection effectiveness of the metamorphic relations identified by each tester with RT and RTo. In the tests, the null hypotheses (H_0) were that the group of metamorphic relations identified by an individual tester had similar fault-detection effectiveness to RT/RTo; the significance level was set to 0.05.

As can be observed from Table VI and Fig. 3, it is statistically significant that the metamorphic relations identified by a single tester detected more faults than RT. However, we could not obtain a general conclusion whether the metamorphic relations identified by a single tester were as effective as RTo with respect to the number of revealed faults. For two programs (MultipleKnapsack and SparseMatrixMultiply), there was no statistically significant difference between the fault-detection effectiveness of RTo and the metamorphic relations identified by a single tester; for the remaining three programs, it was statistically significant that RTo had higher fault-detection effectiveness than all the metamorphic relations identified by a single tester.

A histogram analysis was again used to quantitatively compare the fault-detection capabilities of the metamorphic relations identified by a single tester and a test oracle (refer to Table IV and related discussion in Section VI-B.1 for details of such a method and the grouping scheme). The grouping of the fault-detection effectiveness of testers is summarized in Table VII, where the number of testers in each group for each subject program is given. For example, the value of "2" in the entry corresponding to Group 9 and the program FindKNN means that there were two testers who individually identified a group of metamorphic relations for FindKNN that outperformed RT by 80% to 90% of the difference in effectiveness between RT and RTo.

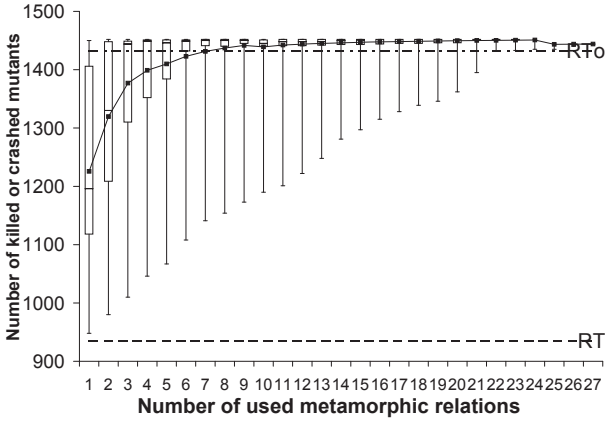
Table VII shows that, on average, 58.82% $(30/51)$ of testers identified metamorphic relations that outperformed RT by at least 90% of the difference in effectiveness between RT and RTo (that is, Group 10). 94.12% $((3+5+2+8+30)/51$, from Groups 6 to 10) of testers were able to identify metamorphic relations that achieved a fault-detection effectiveness half way between that of RT and that of RTo.

In summary, although each tester was a student without training or experience in metamorphic testing prior to this experiment, and was asked to identify metamorphic relations in an independent and ad hoc way, the majority of these students performed well (achieving a fault-detection effectiveness at least half way between that of RT and that of RTo), and over half of them could use metamorphic testing to achieve a very high fault-detection effectiveness (improving on RT by at least 90% of the difference between RT and RTo). Nevertheless, no single tester applying

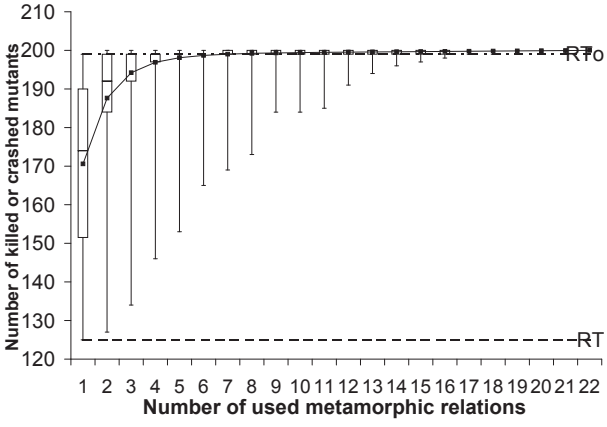


(a) FindKNN

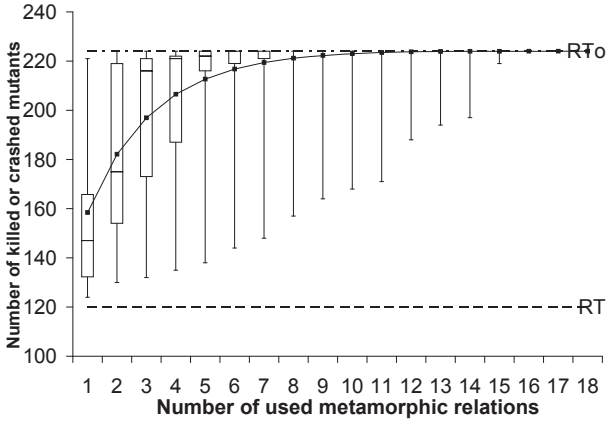
(b) MinimizeDFA



(c) MultipleKnapsack



(d) SparseMatrixMultiply



(e) SetCover

Fig. 2. Relationship between the number of metamorphic relations used and the number of killed/crashed mutants

metamorphic testing could guarantee that a sufficient number of metamorphic relations were identified to imitate an oracle: A testing team composed of different testers was required, as is discussed in the following.

2) Capability of testing teams: In the experiments, every subject program was investigated by two testing teams, each of which consisted of four to seven students from the same university. Fig. 4 reports the fault-detection effectiveness of metamorphic relations with respect to the testing teams, and Table VIII compares this fault-detection effectiveness with that of RT and that of RTo.

As can be observed from Table VIII and Fig. 4, each testing team always identified a set of metamorphic relations that were altogether more effective than RT at revealing faults. On the other hand, generally speaking, there was no significant difference between the fault-detection effectiveness of RTo and the metamorphic relations identified by a testing team. In other words, the set of metamorphic relations identified by a testing team effectively imitates a test oracle, in spite of the fact that the team consisted

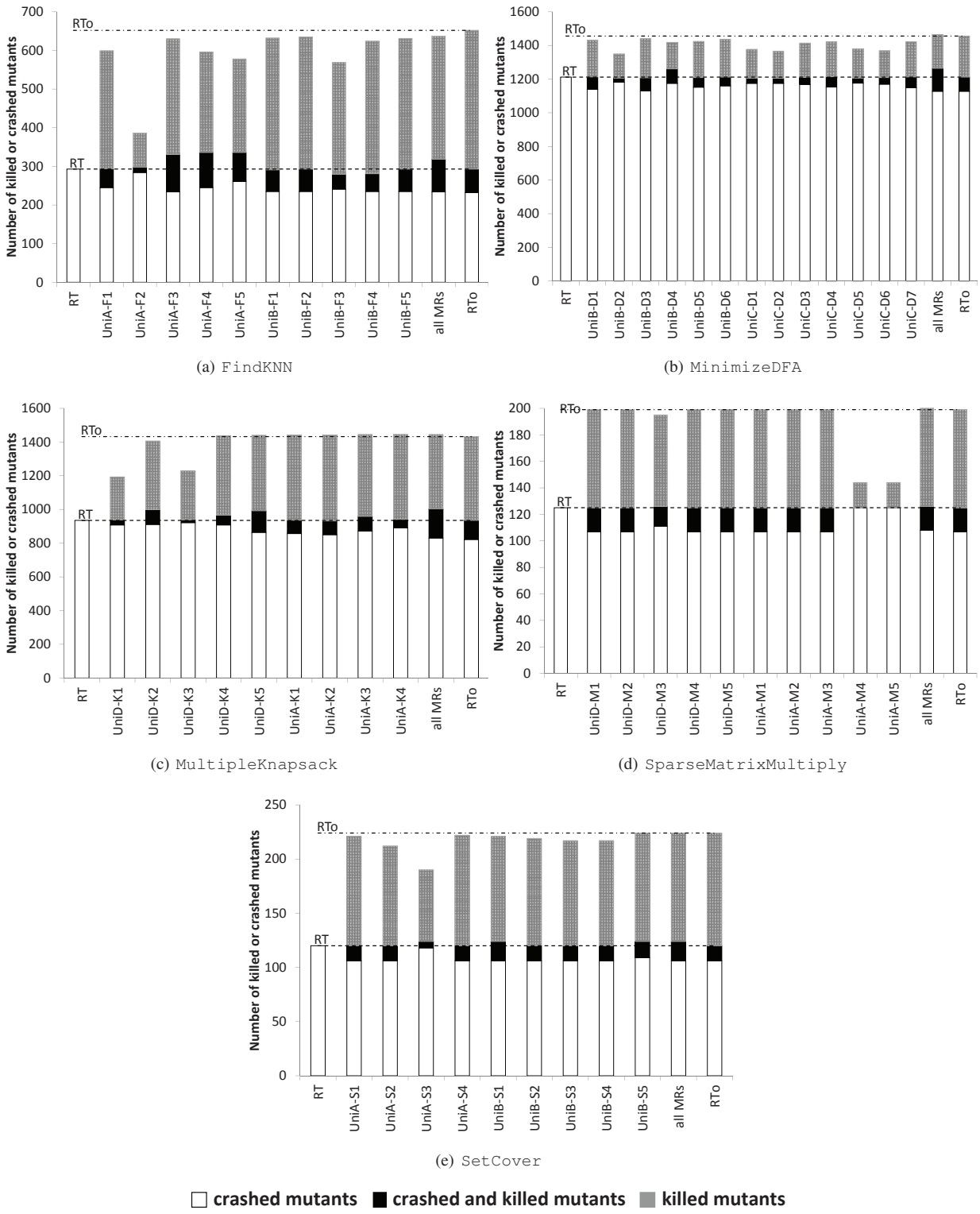


Fig. 3. Relationship between metamorphic relations identified by the same tester and the number of killed/crashed mutants

of inexperienced testers who identified the metamorphic relations in an individual and ad hoc way.

3) Relationship between fault-detection effectiveness and the number of testers: Fig. 5 reports the fault-detection effectiveness of metamorphic relations identified by a group of testers.

Based on the data in Fig. 5, it is possible to calculate the average number of testers (n_{tr}) required to identify a sufficient number of metamorphic relations to outperform RT by at least 90% of the difference in effectiveness between RT and RTo. It was found that $n_{tr} = 2, 3, 1, 2,$ and 1, for `FindKNN`, `MinimizeDFA`, `MultipleKnapsack`, `SparseMatrixMultiply`, and `SetCover`, respectively. These results imply that if three testers were involved, then the metamorphic testing in this case could have been as effective

TABLE VI
T-TESTS FOR COMPARING THE METAMORPHIC RELATIONS IDENTIFIED BY A SINGLE TESTER WITH RT AND RTO

Program	Decision on the comparison with	
	RT	RTo
`FindKNN`	REJECT (5.61×10^{-7})	REJECT (0.0243)
`MinimizeDFA`	REJECT (3.86×10^{-11})	REJECT (5.50×10^{-5})
`MultipleKnapsack`	REJECT (8.87×10^{-7})	ACCEPT (0.2113)
`SparseMatrixMultiply`	REJECT (1.50×10^{-5})	ACCEPT (0.2172)
`SetCover`	REJECT (2.97×10^{-9})	REJECT (0.0462)

H_0: The group of metamorphic relations identified by the same individual tester had similar fault-detection effectiveness to RT/RTo.

TABLE VII
GROUPING OF TESTERS AS COMPARED WITH RT AND RTO

Group	`FindKNN`	`Minimize DFA`	`Multiple Knapsack`	`SparseMatrix Multiply`	`Set Cover`	Total number of testers in a group
1	0	0	0	0	0	0
2	0	0	0	0	0	0
3	1	0	0	2	0	3
4	0	0	0	0	0	0
5	0	0	0	0	0	0
6	0	1	2	0	0	3
7	0	4	0	0	1	5
8	2	0	0	0	0	2
9	2	5	0	0	1	8
10	5	3	7	8	7	30
Total number of testers for a program	10	13	9	10	9	51

TABLE VIII
FAULT-DETECTION EFFECTIVENESS FOR METAMORPHIC RELATIONS IDENTIFIED BY THE TESTING TEAMS COMPARED WITH RTO/RT

Program	Team	M_{oim} when using metamorphic relations identified by a testing team
`FindKNN`	University A	0.9582
	University B	0.95
`MinimizeDFA`	University B	1.0287
	University C	0.9549
`MultipleKnapsack`	University D	1.0101
	University A	1.0262
`SparseMatrixMultiply`	University D	1.1757
	University A	1
`SetCover`	University A	1
	University B	1

as RTo. In other words, only a small number of testers were sufficient to identify metamorphic relations acting as a test result verification mechanism which was as effective as a test oracle, even though the metamorphic relations were identified in an ad hoc way by inexperienced testers.

D. RQ3: Enhancement of metamorphic testing cost-effectiveness

Further investigations revealed that the groups of metamorphic relations with the smallest number of killed or crashed mutants displayed some degree of "similarity": These metamorphic relations appeared to be related to the same properties or characteristics of the subject program. These observations motivated the question of how to enhance the cost-effectiveness of metamorphic testing, and led to the conjecture that relations having more "diversity" (less similarity) would be more cost-effective. Intuitively, for the same number of metamorphic relations, more diverse relations could deliver higher fault-detection effectiveness than similar relations; likewise, comparable fault-detection effectiveness could still be achieved using fewer, but more diverse, metamorphic relations. Therefore, diversity in metamorphic relations should enhance the cost-effectiveness of metamorphic testing.

Experiments were conducted to validate this conjecture. Four assessors with experience in metamorphic testing were recruited to group the metamorphic relations identified in the previous

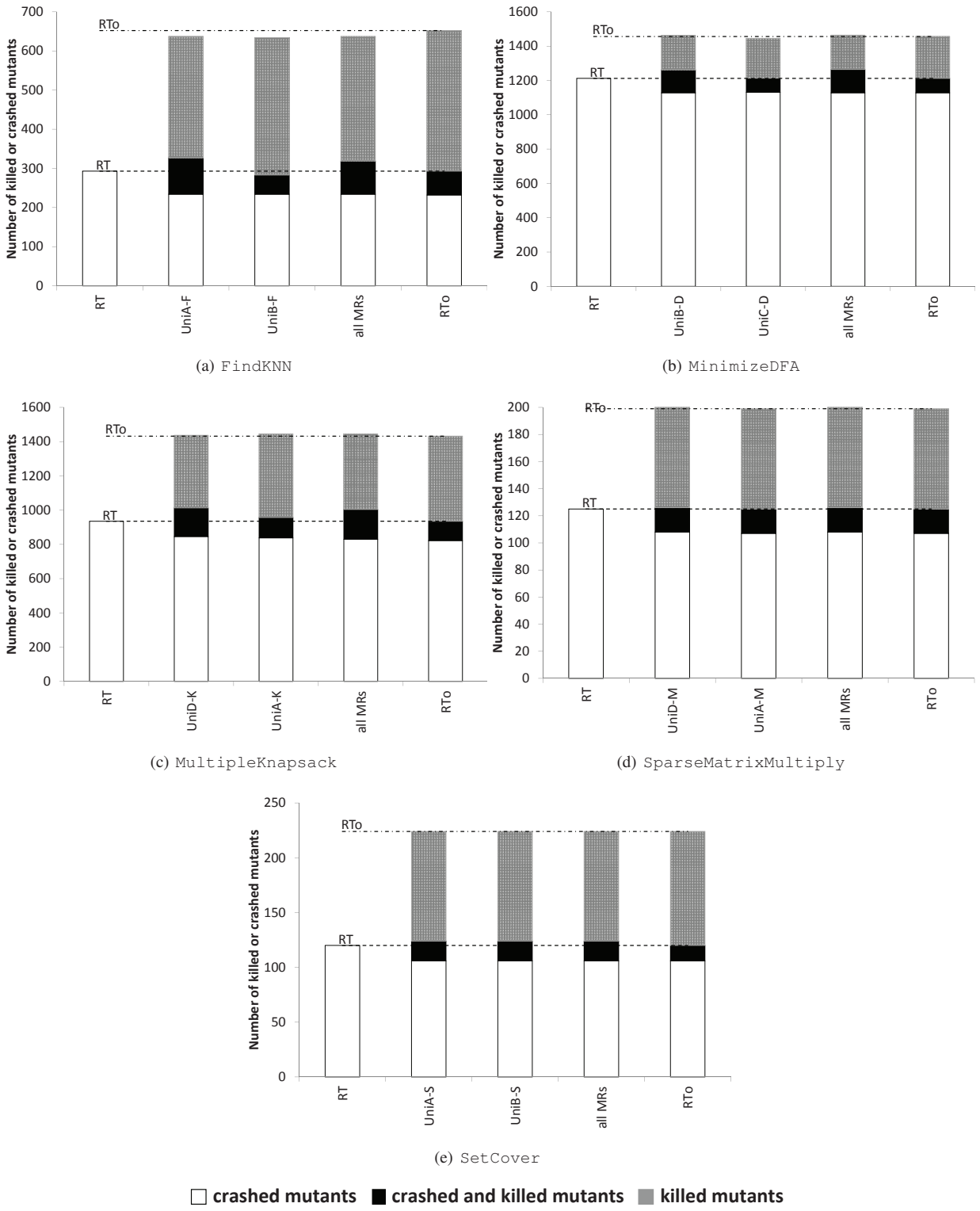
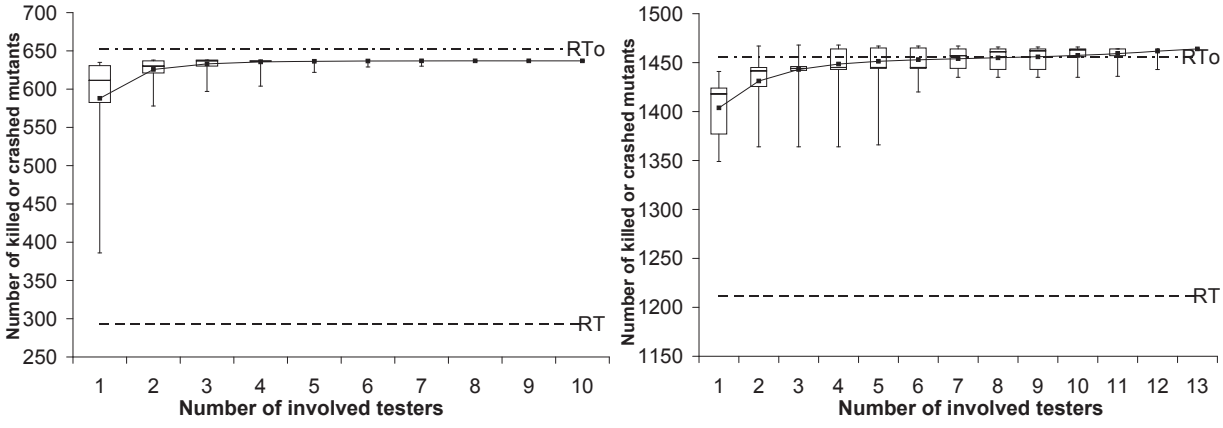


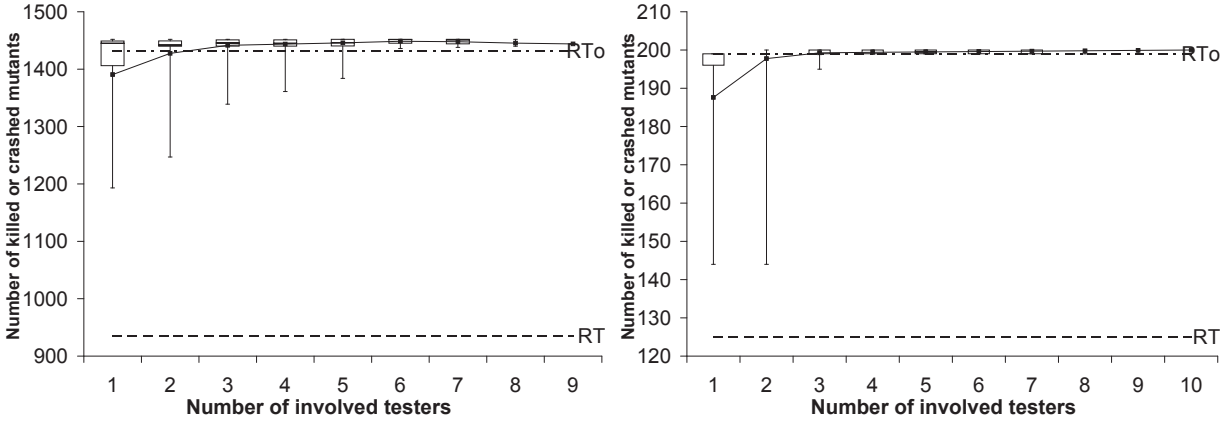
Fig. 4. Relationship between metamorphic relations identified by the same testing team and the number of killed/crashed mutants

experiments based on their own ideas of similarity. As expected, Table IX shows that different assessors had different grouping outcomes, reflecting their different interpretations of similarity.

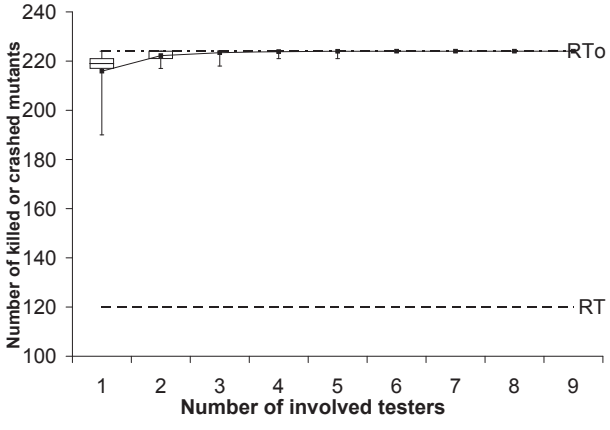
The effectiveness of an assessor's grouping was analyzed as follows. Suppose that an assessor had classified all metamorphic relations for the same subject program into g groups. From these g groups, m ($m = 2, 3, \cdots, g$) groups were randomly chosen, and from each of these m groups, one and only one metamorphic relation was then randomly selected. Intuitively speaking, such m selected metamorphic relations exhibit some kind of diversity because they have been selected from different groups of similar metamorphic relations. The fault-detection effectiveness of these m "diverse" metamorphic relations was then compared with that of m randomly sampled metamorphic relations, which was



(a) `FindKNN` (b) `MinimizeDFA`



(c) `MultipleKnapsack` (d) `SparseMatrixMultiply`



(e) `SetCover`

Fig. 5. Relationship between the number of testers involved and the number of killed/crashed mutants

reported in Section VI-B.3 and Fig. 2. The comparisons were conducted from two perspectives: the average number of killed or crashed mutants (*average effectiveness*); and the standard deviation of the number of killed or crashed mutants (*effectiveness reliability*). Note that we removed redundant metamorphic relations prior to the random sampling: The random sampling was on a pool of distinct metamorphic relations.

The massive amount of data used to derive the average effectiveness and effectiveness reliability of diverse metamorphic relations will not be reported here, but Tables X and XI summarize the two-tailed t-test results for the comparisons. In the tables, the rightmost bottom cell presents the t-test result for all programs and all assessors; each cell in the rightmost column shows the t-test result for each program for all assessors; each cell in the bottom row reports the t-test result for each assessor for all programs; and each of the remaining cells has the t-test result for each combination of program and assessor. The null hypotheses for these t-tests were that m diverse metamorphic relations had

TABLE IX
GROUPING OF METAMORPHIC RELATIONS ACCORDING TO THE ASSESSOR'S OWN INTUITION OF SIMILARITY

Program	Number of metamorphic relations	Number of groups classified by			
		Assessor1	Assessor2	Assessor3	Assessor4
`FindKNN`	16	4	6	10	7
`MinimizeDFA`	16	4	7	13	9
`MultipleKnapsack`	27	9	12	17	12
`SparseMatrixMultiply`	22	6	8	13	10
`SetCover`	18	7	12	13	8

a similar performance to m randomly sampled metamorphic relations in terms of the average effectiveness (Tables X) and the effectiveness reliability (Tables XI). In each cell of the tables, the number in parentheses represents the p-value, based on which the decision (reject or accept) was made.

Based on the experimental data and the t-test results in Tables X and XI, it can be observed that when considering all programs and all assessors, it was statistically significant that both the average effectiveness and the average effectiveness reliability were enhanced by the use of diverse metamorphic relations. 14 of the 20 different assessor–program scenarios had statistically significant higher fault-detection effectiveness and reliability for m diverse metamorphic relations than m randomly sampled metamorphic relations. Only in one scenario (Assessor1 for program `SparseMatrixMultiply`), was there no statistically significant difference for either the average effectiveness or the effectiveness reliability.

With respect to individual subject programs, the t-test results show that the average effectiveness could not be significantly enhanced by more diverse metamorphic relations for two programs (`FindKNN` and `MultipleKnapsack`, as shown in the rightmost column of Table X), but more diverse metamorphic relations could result in more statistically reliable fault-detection effectiveness for all five subject programs (as shown in the rightmost column of Table XI). Furthermore, although the different assessors had different intuitions regarding similarity and diversity, their groupings helped to select diverse metamorphic relations which significantly enhanced both the average effectiveness and the effectiveness reliability (as shown in the bottom rows of Tables X and XI).

VII. THREATS TO VALIDITY

The threats to validity of this study are discussed as follows.

The threat to internal validity is mainly related to the implementation of metamorphic testing, and the generation of (pseudo) random test cases. The programming required was relatively small-scale, and all the source code was carefully reviewed, several times. We are confident that both the metamorphic testing and the random test case generation have been correctly implemented in the experiments.

The major potential threats to external validity relate to the selection of subject programs and the identification of their associated metamorphic relations. As mentioned in Section V-A, due to the experimental constraints, we selected five subject programs of the algorithmic type, and these programs could neither be too complex, nor require much specific domain knowledge — therefore it might be argued that the findings of this study cannot be generalized to any type of program. Nevertheless, we believe that our results are still very useful for providing guidelines for the application of metamorphic testing in practice. Metamorphic testing has been successfully used in the testing of different types of programs, such as online ATM [39], telecommunications [8], wireless metering [25], compilers [40], and office applications [21], [42]. In these previous studies, it has been consistently shown that testers with adequate domain knowledge could identify a sufficient number of metamorphic relations. Compared with the subject programs in this study (each of which implemented a single functionality), simpler programs may need even fewer metamorphic relations to effectively alleviate the oracle problem; similarly, more complicated systems, with multiple distinct functionalities, may require more metamorphic relations to effectively imitate a test oracle. Moreover, since each tester identified metamorphic relations in an independent and ad hoc manner, such a process was somewhat subjective. Several invalid metamorphic relations were also generated, and the number of these might vary with application domains. The recruited testers were university students, who had neither prior knowledge of metamorphic testing nor formal experience in software testing. Nevertheless, if these testers could deliver such promising results after a brief training, it is very likely that more professional testers would be able to identify even more diverse and effective metamorphic relations. In our study, four assessors with experience in metamorphic testing were recruited to classify the identified metamorphic relations into similar groups. Although such a grouping process was subjective and dependent on the assessors' individual understanding of similarity, it did significantly improve the effectiveness of metamorphic testing.

The main potential threat to construct validity is in the measurements used in this study. Fault-detection effectiveness was evaluated based on the number of killed or crashed mutants, a measurement metric commonly used in experiments using mutation analysis, which has been acknowledged as a popular and fair method for evaluating a testing method's effectiveness. In addition, we introduced an oracle imitation measure to quantitatively examine the extent to which metamorphic testing can imitate the fault-detection effectiveness of a test oracle. This measure compares metamorphic testing's fault-detection with that of random testing with and without an oracle, giving the relative degree to which metamorphic testing approximates the oracle.

There should be little threat to the conclusion validity in this study: A large number of test cases were used in the testing, and the experiments resulted in a huge amount of data, which enabled a statistically reliable conclusion. Furthermore, formal statistical tests were employed to verify the statistical significance of the experimental results.

TABLE X
COMPARISON OF THE AVERAGE EFFECTIVENESS OF DIVERSE AND RANDOMLY SAMPLED METAMORPHIC RELATIONS

Program	Assessor1	Assessor2	Assessor3	Assessor4	All assessors
`FindKNN`	REJECT (0.0086)	REJECT (5.33×10^{-5})	REJECT (0.0381)	ACCEPT (0.2639)	ACCEPT (0.2256)
`MinimizeDFA`	REJECT (0.0406)	REJECT (0.0087)	REJECT (0.0176)	REJECT (4.63×10^{-6})	REJECT (7.18×10^{-5})
`Multiple Knapsack`	REJECT (0.0005)	REJECT (2.16×10^{-7})	ACCEPT (0.9078)	REJECT (0.0411)	ACCEPT (0.5590)
`SparseMatrix Multiply`	ACCEPT (0.4204)	REJECT (0.0002)	ACCEPT (0.8724)	REJECT (0.0137)	REJECT (0.0364)
`SetCover`	REJECT (0.0005)	REJECT (2.73×10^{-6})	REJECT (0.0003)	REJECT (0.0002)	REJECT (1.18×10^{-7})
All programs	REJECT (0.0431)	REJECT (0.0353)	REJECT (0.0024)	REJECT (0.0061)	REJECT (0.0025)

H_0: m diverse metamorphic relations had similar average effectiveness to m randomly sampled metamorphic relations.

TABLE XI
COMPARING THE EFFECTIVENESS RELIABILITY OF DIVERSE AND RANDOMLY SAMPLED METAMORPHIC RELATIONS

Program	Assessor1	Assessor2	Assessor3	Assessor4	All assessors
`FindKNN`	REJECT (0.0022)	REJECT (0.0061)	ACCEPT (0.2429)	REJECT (0.0084)	REJECT (0.0035)
`MinimizeDFA`	REJECT (0.0271)	REJECT (0.0050)	ACCEPT (0.5845)	REJECT (0.0022)	REJECT (8.98×10^{-5})
`Multiple Knapsack`	REJECT (0.0018)	REJECT (0.0311)	REJECT (5.90×10^{-6})	REJECT (0.0002)	REJECT (0.0081)
`SparseMatrix Multiply`	ACCEPT (0.5212)	REJECT (2.73×10^{-5})	REJECT (0.0080)	REJECT (0.0017)	REJECT (0.0001)
`SetCover`	REJECT (0.0097)	REJECT (0.0002)	REJECT (4.51×10^{-5})	REJECT (0.0039)	REJECT (1.23×10^{-7})
All programs	REJECT (0.0252)	REJECT (0.0020)	REJECT (2.70×10^{-7})	REJECT (8.14×10^{-7})	REJECT (2.99×10^{-10})

H_0: m diverse metamorphic relations had similar effectiveness reliability to m randomly sampled metamorphic relations.

VIII. DISCUSSION AND CONCLUSION

Metamorphic testing is an approach to software testing which can alleviate the oracle problem. It makes use of some necessary properties (metamorphic relations) of the software under test to provide a test result verification mechanism which can imitate a test oracle. This paper has presented empirical evidence to support this approach, including providing answers to the following questions: to what extent can metamorphic testing alleviate the oracle problem; how easily and successfully can testers detect faults using metamorphic testing; and what are the key factors that influence the effectiveness of metamorphic testing?

In the presented study, several groups of undergraduate and postgraduate students were recruited to identify metamorphic relations in five subject programs of algorithmic type. Even though the metamorphic relations were identified in an individual, independent, and ad hoc manner, by students who had neither formal testing experience nor prior knowledge of metamorphic testing, the identified metamorphic relations had very high fault-detection effectiveness. The fault-detection effectiveness and the average number of metamorphic relations identified by an individual tester are consistent with results observed in independent studies involving subject programs from different application domains [21], [42]. In the experiments, almost every identified metamorphic relation (except MR13 for `SparseMatrixMultiply` and MR10 for `FindKNN`) was able to detect more faults than the commonly adopted approach of crashing. It was observed that for each program, the aggregate of all its identified metamorphic relations could reveal a similar number of faults to a test oracle, which is the base program in this study. Further investigation revealed that the cost-effectiveness of the approach could be improved by reducing the number of metamorphic relations used: It was found that an average of three to six diverse metamorphic relations were sufficient to achieve comparable fault-detection effectiveness to a test oracle.

Although it was initially surprising that a small number of diverse metamorphic relations were sufficient to match the fault-detection effectiveness of a test oracle, a reflection shows that it is in fact intuitively appealing. Consider the following simple example, which explains the underlying rationale for why several metamorphic relations may be able to imitate a test oracle. Suppose a program P accepts a real number x, and outputs the value of a polynomial of degree n, $f(x) = \sum_{i=0}^{n} a_i x^i$. According to the competent programmer hypothesis, a faulty program should not be very different from the correct implementation [12]. Technically speaking, even if P is faulty, it would most likely output the value of a similar function, for example, another polynomial of degree m, $g(x) = \sum_{i=0}^{m} b_i x^i$. Define $h(x) = f(x) - g(x) = \sum_{i=0}^{\max(m,n)} c_i x^i$, where $c_i = \begin{cases} a_i - b_i & \text{if } i \leq \min(m,n), \\ a_i & \text{if } m < i \leq n, \\ -b_i & \text{if } n < i \leq m. \end{cases}$ Since h is a polynomial

of, at most, degree $\max(m,n)$, there are at most $\max(m,n)$ roots for the equation $h(x)=0$. In other words, if P is faulty, there are at most $\max(m,n)$ values of x for which $f(x)$ and $g(x)$ give the same value (that is, $f(x)=g(x)$). Consequently, any set of $(\max(m,n)+1)$ $(x,f(x))$ pairs is sufficient to verify whether P is implementing f or g. Suppose that there are a total of N possible inputs for P (that is, possible values of x). Normally, $N \gg \max(m,n)$; in theory, N may be infinite. If P implements g instead of f, then the probability of selecting a value of x such that $g(x)=f(x)$ is very small, that is, $Prob\,(g(x)=f(x)) \leq \frac{\max(m,n)}{N} \ll 1$. Given k $(k < \max(m,n))$ arbitrarily selected values for x, $Prob\,(g(x)\neq f(x)$ for at least one $x) \geq \left(1-\prod_{j=0}^{k-1}\frac{\max(m,n)-j}{N-j}\right)$. Since $\frac{\max(m,n)-j}{N-j}$ is very small, a small k is already enough to bring $\left(1-\prod_{j=0}^{k-1}\frac{\max(m,n)-j}{N-j}\right)$ close to 1. In other words, even if testing involves a small number of values for x, it is very likely to reveal that the implementation is actually for g instead of the intended f. In this example, $(\max(m,n)+1)$ $(x,f(x))$ pairs collectively serve as a test oracle to distinguish f and g; and each individual $(x,f(x))$ pair is a necessary condition for the correct implementation of f instead of g, and hence can be considered analogous to a metamorphic relation, which is also a necessary property of the program. Considering such an analogy — that the comparison of k and $(\max(m,n)+1)$ $(x,f(x))$ pairs is similar to the comparison of a few metamorphic relations to a test oracle — we can say that if a number of diverse metamorphic relations hold, it is very likely that the specifications have been correctly implemented. Therefore, it is intuitively appealing that a few diverse metamorphic relations should be able to perform as well as a test oracle in terms of revealing faults in a program.

As found previously [21], [42], this study demonstrated that metamorphic testing is simple in concept and thus easy to understand and use. The potential ease with which testers could apply metamorphic testing was also investigated. Although the recruited testers were students inexperienced in testing, and were given only a small amount of training, most of them could easily identify a sufficient number of metamorphic relations to achieve a fault-detection effectiveness at least half way between that of using a test oracle and that from only crashing; and over half of them could identify metamorphic relations that collectively revealed a similar number of faults to a test oracle. However, in general, an individual tester applying metamorphic testing could not be guaranteed to identify sufficient metamorphic relations to imitate a test oracle. On the other hand, every testing team, consisting of four to seven testers, was able to identify enough metamorphic relations to have a similar fault-detection effectiveness to a test oracle. The experimental results also showed that a testing team could be composed of as few as three testers and still yield a sufficient number of metamorphic relations to achieve comparable fault-detection effectiveness to a test oracle.

Further investigation of the experimental results determined that a critical factor affecting the cost-effectiveness of metamorphic testing was the diversity of the metamorphic relations used. It was found that when the metamorphic relations exhibited a certain degree of diversity, they tended to cover different types of faults, and thus had a high fault-detection effectiveness. As long as they were diverse, a small number of metamorphic relations were as effective as a test oracle in revealing faults. Using this notion of diversity, the cost-effectiveness of metamorphic testing could be significantly improved. This is consistent with observations made in investigations for the second research question (Section VI-C), namely that a testing team composed of several testers was more likely than an individual tester to identify sufficient metamorphic relations to deliver comparable fault-detection effectiveness to a test oracle.

Additionally, the results of the empirical studies also provide important insights into how to best conduct metamorphic testing. First and foremost, the diversity of metamorphic relations has been identified as more important than their quantity. In this study, a small number of diverse metamorphic relations were sufficient to detect most faults. Consequently, a smaller team of testers with diverse backgrounds may be better than a larger team of testers with similar backgrounds, because the former is more likely to identify more diverse metamorphic relations. Moreover, it is strongly recommended that a tester should take diversity into account when selecting metamorphic relations for testing.

All the experimental results consistently showed that metamorphic testing was a simple yet effective approach to alleviating the oracle problem. It is therefore worthwhile to continue research into metamorphic testing. As a first attempt to evaluate how effectively metamorphic relations may be able to approach the fault-finding efficiency of a test oracle, this study used some algorithmic programs as the subjects in our experiments. As shown in previous studies, as long as testers had adequate domain knowledge, it was not difficult to identify sufficient metamorphic relations for various application domains, such as online ATM [39], telecommunications [8], wireless metering [25], compilers [40], and office applications [21], [42]. Large-scale studies in these domains would further demonstrate the general applicability of metamorphic testing in practice. There exist a few other techniques for tackling the oracle problem, such as those mentioned in Section III. It will be interesting to continue our research through comparing metamorphic testing with these techniques. In addition, although metamorphic relations identified in an ad hoc manner may already have a good performance, it is still essential to develop more systematic approaches for their identification: Only when metamorphic relations can be systematically identified, can the performance of metamorphic testing become more predictable and controllable. With a systematic approach, it should also become possible to provide testers with better training, and thus achieve better testing results. Another important research direction relates to the concept and interpretation of diversity for metamorphic relations. Although diversity has been widely used in test case selection [9], [19], its application to metamorphic relations, as investigated in this paper, is relatively abstract and subjective, and strongly dependent on the experience and background of the testers. It will be interesting to develop a more concrete concept of diversity for metamorphic testing, as has already been done in the area of test case selection, and apply it to the identification or selection of metamorphic relations. We are strongly confident that such investigations will further enhance the cost-effectiveness of metamorphic testing and software testing.

ACKNOWLEDGMENTS

We are grateful to Kai-Yuan Cai of Beihang University, Changai Sun of University of Science and Technology Beijing, Zhenyu Chen of Nanjing University, and Jianjun Zhao of Shanghai Jiao

Tong University for their help in facilitating the experiments. We are also thankful to the students from these universities who participated in the experiments.

REFERENCES

[1] P. E. Ammann and J. C. Knight. Data diversity: An approach to software fault tolerance. *IEEE Transactions on Computers*, 37(4):418–425, 1988.

[2] A. Avizienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[3] A. C. Barus, T. Y. Chen, D. Grant, F.-C. Kuo, and M.-F. Lau. Testing of heuristic methods: A case study of greedy algorithm. In *Proceedings of the 3rd IFIP TC 2 Central and Eastern European Conference on Software Engineering Techniques (CEE-SET 2008)*, volume 4980 of *Lecture Notes in Computer Science*, pages 246–260, 2011.

[4] S. Beydeda. Self-metamorphic-testing components. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)*, pages 265–272, 2006.

[5] T. Y. Chen, S. C. Cheung, and S. M. Yiu. 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, 1998.

[6] T. Y. Chen, J. W. K. Ho, H. Liu, and X. Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10:24:1–24:12, 2009.

[7] T. Y. Chen, D. H. Huang, T. H. Tse, and Z. Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JIISIC 2004)*, pages 569–583, 2004.

[8] T. Y. Chen, F.-C. Kuo, H. Liu, and S. Wang. Conformance testing of network simulators based on metamorphic testing technique. In *Proceedings of the 29th IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2009)*, volume 5522 of *Lecture Notes in Computer Science*, pages 243–248, 2009.

[9] T. Y. Chen, F.-C. Kuo, R. G. Merkel, and T. H. Tse. Adaptive random testing: The ART of test case diversity. *Journal of Systems and Software*, 83(1):60–66, 2010.

[10] T. Y. Chen, T. H. Tse, and Z. Zhou. Semi-proving: An integrated method for program proving, testing, and debugging. *IEEE Transactions on Software Engineering*, 37(1):109–125, 2011.

[11] T. Y. Chen and Y. T. Yu. On the relationship between partition and random testing. *IEEE Transactions on Software Engineering*, 20(12):977–980, 1994.

[12] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, 1978.

[13] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4):405–435, 2005.

[14] X. Feng, D. L. Parnas, T. H. Tse, and T. O'Callaghan. A comparison of tabular expression-based testing strategies. *IEEE Transactions on Software Engineering*, 37(5):616–634, 2011.

[15] J. E. Forrester and B. P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *Proceedings of the 4th USENIX Windows Systems Symposium (USENIX-WIN 2000)*, pages 59–68, 2000.

[16] G. Fraser and A. Zeller. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th International Symposium on Software Testing and Analysis (ISSTA 2010)*, pages 147–158, 2010.

[17] A. Gotlieb and B. Botella. Automated metamorphic testing. In *Proceedings of the 27th Annual International Computer Software and Applications Conference (COMPSAC 2003)*, pages 34–40, 2003.

[18] B. Hailpern and P. Santhanam. Software debugging, testing, and verification. *IBM Systems Journal*, 41(1):4–12, 2002.

[19] H. Hemmati, A. Arcuri, and L. Briand. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology*, 22(1):6:1–6:42, 2013.

[20] R. M. Hierons. Oracles for distributed testing. *IEEE Transactions on Software Engineering*, 38(3):629–641, 2012.

[21] P. Hu, Z. Zhang, W. K. Chan, and T. H. Tse. An empirical comparison between direct and indirect test result checking approaches. In *Proceedings of the 3rd International Workshop on Software Quality Assurance (SOQUA 2006)*, pages 6–13, 2006.

[22] Y. F. Hu, R. J. Allan, and K. C. F. Maguire. Comparing the performance of JAVA with Fortran and C for numerical computing. Technical report, Daresbury Laboratory, CCLRC, 2000. http://www.dl.ac.uk/TCSC/UKHEC/JASPA/bench.pdf.

[23] JFlex. The fast scanner generator for java. http://jflex.de/, 2009.

[24] J. C. Knight and N. G. Leveson. An experimental evaluation of the assumption of independence in multi-version programming. *IEEE Transactions on Software Engineering*, 12(1):96–109, 1986.

[25] F.-C. Kuo, T. Y. Chen, and W. K. Tam. Testing embedded software by metamorphic testing: A wireless metering system case study. In *Proceedings of the 36th IEEE Conference on Local Computer Networks (LCN 2011)*, pages 295–298, 2011.

[26] H. T. Lau. *A Java Library of Graph Algorithms and Optimisation*. Taylor & Francis Group, Boca Raton, 2007.

[27] Y.-S. Ma, J. Offutt, and Y.-R. Kwon. MuJava : An automated class mutation system. *Software Testing, Verification and Reliability*, 15(2):97–133, 2005.

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

[29] J. Mayer and R. Guderlei. An empirical study on the selection of good metamorphic relations. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMPSAC 2006)*, pages 475–484, 2006.

[30] B. P. Miller, G. Cooksey, and F. Moore. An empirical study of the robustness of MacOS applications using random testing. *ACM SIGOPS Operating Systems Review*, 41(1):78–86, 2007.

[31] B. P. Miller, L. Fredriksen, and B. So. An empirical study of the reliability of UNIX utilities. *Communications of the ACM*, 33(12):32–44, 1990.

[32] C. Murphy, K. Shen, and G. E. Kaiser. Automatic system testing of programs without test oracles. In *Proceedings of the 18th International Symposium on Software Testing and Analysis (ISSTA 2009)*, pages 189–200, 2009.

[33] G. J. Myers. *The Art of Software Testing*. John Wiley and Sons, second edition, 2004. Revised and updated by T. Badgett and T. M. Thomas with C. Sandler.

[34] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*. Cambridge University Press, 1992.

[35] P. Rao, Z. Zheng, T. Y. Chen, N. Wang, and K. Cai. Impacts of test suites class imbalance on spectrum-based fault localization techniques. In *The Symposium on Engineering Test Harness (TSE-TH 2013)*, in press.

[36] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[37] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53(3):245–258, 2011.

[38] M. Staats, G. Gay, and M. P. E. Heimdahl. Automated oracle creation support, or: How I learned to stop worrying about fault propagation and love mutation testing. In *Proceedings of the 34th International Conference on Software Engineering (ICSE 2012)*, pages 870–880, 2012.

[39] C. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T. Y. Chen. Metamorphic testing for web services: Framework and a case study. In *Proceedings of the 9th International Conference on Web Services (ICWS 2010)*, pages 283–290, 2011.

[40] Q. Tao, W. Wu, C. Zhao, and W. Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *Proceedings of the 17th Asia Pacific Software Engineering Conference (APSEC 2010)*, pages 270–279, 2010.

[41] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu. Metamorphic slice: An application in spectrum-based fault localization. *Information and Software Technology*, 55(5):866–879, 2013.

[42] Z. Zhang, W. K. Chan, T. H. Tse, and P. Hu. Experimental study to compare the use of metamorphic testing and assertion checking. *Journal of Software*, 20(10):2637–2654, 2009.

Huai Liu is a Research Fellow at the Australia-India Centre for Automation Software Engineering, RMIT University, Australia. He received his B.Eng. in physioelectronic technology and M.Eng. in communications and information systems, both from Nankai University, China; and PhD degree in software engineering from Swinburne University of Technology, Australia. His current research interests include software testing, cloud computing, and end-user software engineering.

Fei-Ching Kuo is a Senior Lecturer at the Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia. She received her Bachelor of Science Honours in Computer Science and PhD in Software Engineering, both from Swinburne University of Technology, Australia. She was a lecturer at University of Wollongong, Australia. She is also the Program Committee Chair for the 10th International Conference on Quality Software 2010 (QSIC'10) and Guest Editor of a Special Issue for the Journal of Systems and Software, special issue for Software Practice and Experience, and special issue for International Journal of Software Engineering and Knowledge Engineering. Her current research interests include software analysis, testing and debugging.

Dave Towey is an assistant professor in the Division of Computer Science, The University Nottingham Ningbo China, prior to which he was with Beijing Normal University–Hong Kong Baptist University: United International College, China. His background includes an education in computer science, linguistics and languages (BA/MA from the University of Dublin, Trinity College), and PhD in computer science from the University of Hong Kong. His research interests include software testing, software design, and technology-enhanced education. He is a member of both the IEEE and the ACM.

Tsong Yueh Chen is a Professor of Software Engineering at the Faculty of Information and Communication Technologies, Swinburne University of Technology, Australia. He received his BSc. and MPhil. from The University of Hong Kong; MSc and DIC from Imperial College of Science and Technology; and PhD degree from The University of Melbourne. His current research interests include software testing, debugging, software maintenance, and software design.