HKU CS Tech Report TR-2009-01

To appear in *Journal of Software*

An Experimental Study to Compare the Use of Metamorphic Testing and Assertion Checking* **

ZHANG Zhenyu[1], CHAN W. K.[2+], TSE T. H.[3], HU Peifeng[3]

[1](Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong)

[2](Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Hong Kong)

[3](Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong)

[3](China Merchants Bank, Central, Hong Kong)

+ Corresponding author: Phone: +852 2788 9684, Fax: +852 2788 8614, E-mail: wkchan@cs.cityu.edu.hk, http://www.cs.cityu.edu.hk/~wkchan/

Abstract: A test oracle in software testing is a mechanism for checking whether the program under test behaves correctly for any execution. In some practical situations, oracles can be unavailable or too expensive to apply. Metamorphic testing (MT) was proposed to alleviate this problem so that software can be delivered under the time-to-market pressure. However, the effectiveness of MT has not been studied adequately. This paper conducts a controlled experiment to investigate the cost effectiveness of using MT. The fault detection capability and time cost of MT are compared with the standard assertion checking method. Our results show that MT has potentials to detect more faults than the assertion checking method. The experiment results also show a trade-off between the two testing methods: MT can be less efficient but more effective, and can be defined at a coarser level of granularity than the assertion checking method.

Key words: Metamorphic testing, assertion checking, test oracle, controlled experiment, empirical evaluation

1 Introduction

Software testing is a key activity in any software development project. It assures applications by executing programs over test cases with the intent to reveal failures [2]. To conduct testing, software testers usually evaluate the test results through an oracle, which is a mechanism for checking whether a program behaves correctly [3]. Many programs do not have a full specification, and many of them are developed without similar versions for reference. In these situations, oracles may be unavailable or too expensive to apply. This is known as the *test oracle problem* [3]. The oracle problem is not limited to the above kind of scenarios. For instance, for programs involving complex computations (such as partial differential equations [4], graphics-based software [5][6], database applications [7], large-scale components, web server, or operating systems [7]), their outputs are difficult to verify. In current software practices, the oracle is often a human tester who checks the testing results manually. The manual

** This research is supported in part by grants of the Research Grants Council of Hong Kong (project numbers 111107 and 717308) and the Australian Research Council (project number DP0984760). A preliminary version of this paper was presented at the *3rd International Workshop on Software Quality Assurance* (*SOQUA 2006*) in conjunction with the *14th ACM SIGSOFT Symposium on Foundations of Software Engineering* (*SIGSOFT 2006/FSE-14*) [1].

checking of program output acutely limits the efficiency of testing and increases its cost, especially when there is a need to verify the results of a large number of test cases. Assessing the correctness of program outcomes has, therefore, been recognized as "one of the most difficult tasks in software testing" [8].

As we shall review in Section 2, metamorphic testing (MT) [4][9][10][11][12] and assertion checking [13][14] are techniques to alleviate the oracle problem. Assertion checking verifies the test result or intermediate states of the program when executing a test case. It directly confirms the execution behavior of a program in terms of a checking condition of program states or individual outputs. MT takes another direction, which verifies follow-up test cases based on an initial set of test cases. Apart from test case generation, MT also helps verify the relations among the results of these initial test cases and their follow-up test cases. In other words, MT indirectly verifies the behaviors of multiple program executions in terms of a checking condition of (input and output) data. It would be interesting to compare the two approaches on their performance in identifying failures. As an analogy, if we view a test case, its execution, and the output collectively as an entity (as in entity-relationship diagrams), assertion checking verifies the correctness of individual entities, whereas MT further verifies the correctness of the relationships among entities.

To measure the performance of a testing technique, it is popular in academic research to study its effectiveness. However, effectiveness and efficiency are complementary so that they give a proper performance picture of a testing technique. In this paper, we study both dimensions.

There have been various case studies in applying metamorphic testing to different types of programs, ranging from conventional programs and object-oriented programs, to pervasive programs and web services. Chen et al. [4] reported on the testing of programs for solving partial differential equations. They [15] further investigated the integration of metamorphic testing with fault-based testing and global symbolic evaluation. Gotlieb and Botella [16] developed an automated framework to check against a class of metamorphic relations. Chan and colleagues applied metamorphic approach to the unit testing [17] and integration testing [9] of context-sensitive middleware-based applications. Chan and others [11][18] also developed a metamorphic approach to online testing of service-oriented software applications. The improvement on the binary classification approach to alleviate the test oracle problem for graphics-intensive applications has been investigated in [5][6]. Throughout these studies, both the testing and the evaluation of experimental results were conducted by the researchers themselves. There is a need for systematic empirical research on how well MT can be applied in practical and yet generic situations and how effective MT is compared with other testing strategies.

Like other comparisons of testing strategies such as between control flow and data flow test adequacy criteria [19] and among different data flow test adequacy criteria [1], controlled experimental evaluations are essential. They should answer the following research questions: (a) Can testers be trained to apply MT properly? (b) How does the fault detection effectiveness of MT compare with other effective strategies? (c) What is the time cost to apply MT? (d) What is the cost to apply MT if some artifacts of MT implementation are faulty?

In this paper, we report and discuss the results in a controlled experiment setting with a view to answering the above questions. The subject participants were 38 postgraduate students enrolled in an advanced software testing course. They have completed a bachelor degree in computer science or equivalent. Before doing the experiment, they were taught the concepts of MT and a reference strategy (namely, assertion checking [13]) to alleviate the oracle problem. The training sessions for either concept were similar in duration. Three open-source programs were selected as target programs. The subjects were required to apply both MT and assertion checking strategies to test these programs independently. We ran their test cases over a representative set of faulty versions of the target programs to assess the capability of these two testing strategies in detecting faults [5][20]. The raw data were analyzed with a view to comparing the costs and effectiveness between MT and assertion checking. We further ran test cases having faulty metamorphic relations over faulty versions of the target programs to assess whether faulty metamorphic relations may seriously affect the effectiveness of applying MT.

The main contribution of this paper is six-fold: (i) It is the first controlled experiment to compare metamorphic testing and assertion checking. (ii) The experiment shows that metamorphic testing is more effective than assertion checking as a means to identify faults. (iii) It provides empirical evidence to resolve the speculation whether subjects have difficulty formulating metamorphic relations and implementing MT. Indeed, the results of the experiment show that all subjects manage to propose metamorphic relations for the target programs after a brief general introduction on MT, and identical or very similar metamorphic relations are proposed by different subjects. (iv) It shows that there is a tradeoff between metamorphic testing and assertion checking when applying them to alleviate the test oracle problem. The empirical results indicate that metamorphic testing is worth applying in terms of time cost whereas assertion checking is more efficient to apply. (v) This paper further reports the first experiment to evaluate the effectiveness of (correct and faulty) metamorphic relations in MT. The result shows that a test suite can effectively identify failures from faulty target programs despite the presence of faulty metamorphic relation implementations. (vi) Our analysis on raw data also indicates that the granularity of using MT is coarser than assertion checking in failure detection.

The paper is organized as follows: Section 2 reviews the related literature. Section 3 introduces the fundamental notions and procedures of metamorphic testing. Section 4 describes the controlled experiment, and the result is presented and discussed in Section 5. Finally, Section 6 concludes the paper.

2 Related Work

Many approaches have been proposed to alleviate the test oracle problem. Rather than checking the test output directly, they usually propose to construct various types of oracle variant to verify the correctness of the program under test. Chapman [21] suggested that a previous version of a program could be used to verify the correctness of the current version. It is now a popular practice in regression testing. However, using this approach, testers need to identify whether the test case is applicable to the previous version.

Weyuker [3] suggested checking whether some identity relations would be preserved by the program under test. This notion of equivalence has been well-adopted in practice.

Blum and others [22][23] proposed a program checker, which was an algorithm for checking the output of computation for numerical programs. Their theory was subsequently extended into the theory of self-testing/correcting [24].

Xie and Memon [25] studied different types of oracle for graphic user interface (GUI) testing. Binder [13] discussed four categories and eighteen oracle patterns in object-oriented program testing.

Assertion checking [26] is another method to verify the execution results of programs. An assertion, which is usually embedded directly in the source code of the program under test, is a Boolean expression that verifies whether the execution of a test case satisfies some necessary properties for correct implementation. Assertions are supported by many programming languages and are easy to implement. It has been incorporated in the Microsoft .Net platform. Assertion checking has been widely used in testing. For example, state invariants [13][27], represented by assertions, can be used to check the stated-based behaviors of a system. Briand et al. [28] investigated the effectiveness of using state-invariant assertions as oracles and compared it with the results using precise oracles for object-oriented programs. It was shown that state-invariant assertions were effective in detecting state-related errors. Since our target programs are also object-oriented programs, we have chosen assertion checking as the alternative testing strategy in our experimental comparison. Assertion checking is also popular in unit testing framework such as JUnit, in which verification of the program states or outputs of a test case can be done during or after the test execution.

The design by contract methodology [29] uses contracts to construct reliable software. Contracts, which are made of assertions, take the form of routine pre-conditions, post-conditions, and class invariants coded into the program under test.

Some researchers have proposed to prepare test specifications, either manually or automatically, to alleviate the test oracle problem. Memon et al. [29] assumed that a test specification of internal object interactions was available and used it to identify nonconformance of the execution traces. This type of approach is common in conformance testing for telecommunication protocols. Sun et al. [17] proposed a similar approach to testing the harnesses of applications. Last and colleagues [30][31] trained pattern classifiers to learn the casual input-output relationships of a legacy system. They then used the classifiers as test oracles. Chan et al. [5] further investigated the feasibility of using pattern classification techniques when the test outputs cannot be accurately determined. Podgurski and colleagues [32][33] classified failure reports into categories via classifiers, and then refined the classification with the aim to extract more information to help testers diagnose program failures. Bowring et al. [34] used a progressive approach to train a classifier to ease the test oracle problem in regression testing. Chan et al. [35] used classifiers to identify different types of behaviors related to the synchronization failures of objects in a multimedia application.

Beydeda [36] proposed to use metamorphic testing as a means to improve the testability of program components. Wu [37] observed that follow-up test cases can be initial test cases of the next round, and thus, proposes to apply MT iteratively to utilize metamorphic relations more economically. Chan et al. [6] proposed a methodology to integrate MT with the pattern classification technique. Murphy [38] explored the application of metamorphic testing to support field testing.

3 Preliminaries of Metamorphic Relations and Testing

This section introduces metamorphic testing. As we have discussed in Section 1, metamorphic testing relies on a checking condition that relates multiple test cases and their results in order to reveal failures. Such a checking condition is known as a *metamorphic relation*. In this section, we revisit metamorphic relations and discuss how they can be used in the metamorphic approach to software testing.

3.1 Metamorphic Relations

A *metamorphic relation* (MR) is a relation over a series of distinct inputs and their corresponding results for multiple evaluations of a target function [20]. Consider, for instance, the sine function. We have the following relation: *If* $x_2 = \pi - x_1$, *then* $\sin x_2 = -\sin x_1$. We note from this example that a metamorphic relation consists of two parts. The first part (denoted by r in the definition below) relates x_2 to x_1. The second part (denoted by r') relates the results of the function. If the MR above is not satisfied for some input, we deem that a failure is revealed.

Definition 1 (Metamorphic Relation) [10] *Let* $\langle x_1, x_2, \ldots, x_k\rangle$ *be a series of inputs to a function* f *, where* $k \geq 1$*, and let* $\langle f\,(x_1), f\,(x_2), \ldots, f\,(x_k)\rangle$ *be the corresponding series of results. Suppose* $\langle f\,(x_{i1}), f\,(x_{i2}), \ldots, f\,(x_{im})\rangle$ i*s a subseries, possibly an empty subseries, of* $\langle f\,(x_1), f\,(x_2), \ldots, f\,(x_k)\rangle$*. Let* $\langle x_{k+1}, x_{k+2}, \ldots, x_n\rangle$ *be another series of inputs to* f *, where* $n \geq k + 1$*, and let* $\langle f\,(x_{k+1}), f\,(x_{k+2}), \ldots, f\,(x_n)\rangle$ *be the corresponding series of results. Suppose, further, that there exists relations* r ($x_1, x_2, \ldots, x_k, f\,(x_{i1}), f\,(x_{i2}), \ldots, f\,(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n$) *and* r' ($x_1, x_2, \ldots, x_n, f\,(x_1), f\,(x_2), \ldots, f\,(x_n)$) *such that* r' *must be true whenever* r *is satisfied. Here, r and r' can be any mathematics relation of aforementioned parameters. We say that*

MR = $\{ \langle x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)\rangle$
$\mid r(x_1, x_2, \ldots, x_k, f(x_{i1}), f(x_{i2}), \ldots, f(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n)$
$\rightarrow r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)) \}$

is a metamorphic relation. When there is no ambiguity, we simply write the metamorphic relation as

MR*: If* $r(x_1, x_2, \ldots, x_k, f(x_{i1}), f(x_{i2}), \ldots, f(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n)$
then $r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))$.

Furthermore, $x_1, x_2, \ldots, x_k$ *are known as initial test cases and* $x_{k+1}, x_{k+2}, \ldots, x_n$ *are known as follow-up test cases.*

Similar to assertions in the mathematical sense, metamorphic relations are also necessary properties of the function to be implemented. They can, therefore, be used to detect inconsistencies in a program. They can be any relations involving the inputs and outputs of *two or more* executions of the target program. They may include inequalities, periodicity properties, convergence properties, subsumption relationships, and other properties.

Intuitively, human testers are needed to study the problem domain related to a target program and formulate metamorphic relations accordingly. This is akin to requirements engineering, in which humans instead of automatic requirements engines are necessary for formulating systems requirements. In some domains where the requirements of an implementation are best specified mathematically, metamorphic relations may readily be identified. Is there a systematic methodology guiding testers to formulate metamorphic relations like the methodologies that guide systems analysts to specify requirements? This remains a challenging question. We shall further investigate along this line in the future. We observe that other researchers are also beginning to formulate important properties in the form of specifications to facilitate the verification of system behaviors [19].

3.2 Metamorphic Testing

In practice, if the program is written by a competent programmer, most test cases will be *passed test cases*, which are test cases that do not reveal any failure. These passed test cases have been considered useless in conventional testing. Metamorphic testing (MT) uses information from such passed test cases, which will be referred to as *initial test cases*.

Consider a program p for a target function f in the input domain D. A series of initial test cases $T = \langle t_1, t_2, ..., t_k\rangle$ can be selected according to any test case selection strategy. Executing the program p on T produces outputs $p(t_1), p(t_2), ..., p(t_k)$. When there is a test oracle, the test results can be verified against $f(t_1), f(t_2), ..., f(t_k)$. If these results reveal any failure, testing stops. On the other hand, when there is no test oracle or when no failure is revealed, the metamorphic testing procedure can continue to be applied to automatically generate follow-up test cases $T' = \{t_{k+1}, t_{k+2}, ..., t_n\}$ based on the initial test cases T so that the program can be verified against metamorphic relations.

Definition 2 (Metamorphic Testing) [10] *Let P be an implementation of a target function f. The* metamorphic testing *of the metamorphic relation*

MR*: If* $r(x_1, x_2, \ldots, x_k, f(x_{i1}), f(x_{i2}), \ldots, f(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n)$,
then $r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n))$

involves the following steps: (i) *Given a series of initial test cases* $\langle x_1, x_2, \ldots, x_k\rangle$ *and their respective results* $\langle P(x_1), P(x_2), \ldots, P(x_k)\rangle$*, generate a series of follow-up test cases* $\langle x_{k+1}, x_{k+2}, \ldots, x_n\rangle$ *according to the relation* $r(x_1, x_2, \ldots, x_k, P(x_{i1}), P(x_{i2}), \ldots, P(x_{im}), x_{k+1}, x_{k+2}, \ldots, x_n)$ *over the implementation P.* (ii) *Check the relation* $r'(x_1, x_2, \ldots, x_n, P(x_1), P(x_2), \ldots, P(x_n))$ *over P. If r' is false, then the metamorphic testing of MR reveals a failure.*

3.3 Metamorphic Testing Procedure

Gotlieb and Botella [16] developed an automated framework for a class of metamorphic relations. The framework translates a specification into a constraint logic programming (CLP) program. Test cases can be automatically generated according to the CLP program using a constraint solving approach. Their framework works on a subset of the C language, but it is not clear whether the framework is applicable to test cases involving objects. Since we want to apply MT to object-oriented programs, we adopt the original procedure [39], which is described as follows:

First, testers identify and formulate metamorphic relations MR_1, MR_2, …, MR_n from the target function f. For each metamorphic relation MR_i, testers construct a function gen_i to generate follow-up test cases from the initial test cases. Next, for each metamorphic relation MR_i, testers construct a function ver_i, which will be used to verify whether multiple inputs and the corresponding outputs satisfy MR_i. After that, testers generate a set of initial test cases T according to a preferred test case selection strategy. Finally, for every test case in T, the test driver invokes the function gen_i to generate follow-up test cases and apply the function ver_i to check whether the test cases satisfy the given metamorphic relation MR_i. If a metamorphic relation MR_i is violated by any test case, ver_i reports that an error is found in the program under test.

4 Experiment

This section describes the set up of the controlled experiment. It first formulates the research questions to be investigated and then describes the experimental design and experimental procedure.

4.1 Research Questions

The research questions to be investigated are summarized as follows:

(a) *Can the subjects properly apply MT after training?* Can the subjects identify correct and useful metamorphic relations from target programs? Can the same metamorphic relations be discovered by multiple subjects?

(b) *Is MT an effective testing method?* Does MT have a comparative advantage over other testing strategies such as assertion checking in terms of the number of mutants detected? To address this question, we shall use the standard statistical technique of null hypothesis testing.

Null Hypothesis H_0: There is no significant difference between MT and assertion checking in terms of the number of mutants detected.

Alternative Hypothesis H_1: There is a significant difference between MT and assertion checking in terms of the number of mutants detected.

We aim at applying the standard concept of the p-value in the Mann-Whitney test to find the confidence level that H_0 should be rejected, with a view to supporting our claim that the difference between MT and assertion checking is statistically significant rather than by chance.

(c) What is the effort, in terms of time cost, in applying MT?

(d) If an MR is faulty, what is the cost of applying MT (in terms of the number of mutants detected)?

4.2 Design of Experiment

Our experiment identifies four independent and three dependent variables. The independent variables are *testing strategies*, *subjects*, *target programs*, *faulty versions of target programs*, and *faulty versions of metamorphic relation programs*. The dependent variables are *time cost*, *number of metamorphic relations/assertions*, and *testing effectiveness in terms of mutation detection ratio*. For the variable on testing strategies, we incorporate MT and assertion checking. In the rest of this section, we describe the other three independent variables. Section 5 will analyze the results according to the dependent variables.

Subjects: All the 38 subjects were graduate students in computer science or equivalent who attended the course "Advanced Topics in Software Engineering: Software Testing" at The University of Hong Kong. These students had at least a bachelor degree in computer science, computer engineering, or electronic engineering. The majority of them were part-time MSc students with some industrial experience. The rests were MPhil and PhD students. We controlled that the training sessions of either approach are comparable in duration and in content. The number of subjects used our controlled experiment is similar to those in other software engineering controlled experiments. For instance, the experiments in [40][41] use 44 subjects.

Since differences in software engineering background might affect the students' capability to apply metamorphic testing or assertion checking, we conducted a brief survey prior to the experimentation. The survey asks subjects their experiences in the industrial environment in each of the following four areas: object-oriented design, Java programming, software testing, and assertion checking.

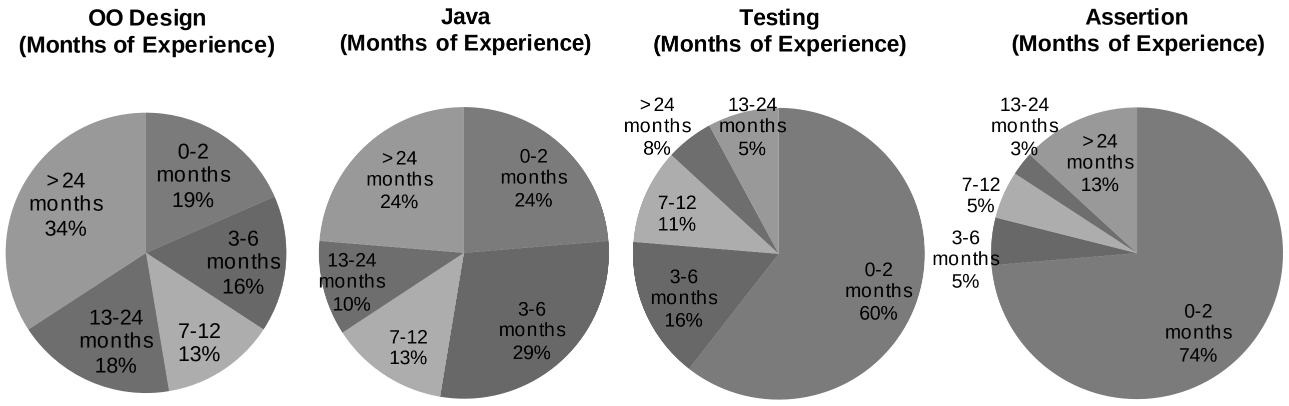

Figure 1 Experiences of Subjects in Object-Oriented Design, Java, Testing, and Assertions

Figure 1 lists the survey result. The overall survey result showed that most of them had real-life or academic experience. As most of subjects were knowledgeable about object-oriented design and Java programming, they were deemed to be competent in the tasks in the controlled experiment. On the other hand, we found a few students having rather limited experience in software testing and assertion checking. Since they did not have prior concepts of metamorphic testing either, the experiment did not specifically favor the metamorphic approach.

Target Programs: We used three open-source programs as target programs. All of them were Java programs selected from real-world software systems.

The first target program Boyer is a program using the Boyer-Moore algorithm to support the applications in Canadian Mind Products, an online commercial software company (available at http://mindprod.com/products1.html). The program returns the index of the first occurrence of a specified pattern within a given text.

The second target program BooleanExpression evaluates Boolean expressions and returns the resulting Boolean values. For example, the program may evaluate the expression "!(true && false) || true" and returns "true". The program is a core part of a popular open-source project jboolexpr (available at http://sourceforge.net/projects/jboolexpr) in SourceForge (URL http://www.sourceforge.net), the largest open-source project website.

The third target program is TxnTableSorter. It is taken from a popular open-source project Eurobudget (available at http://eurobudget.sourceforge.net) in the SourceForge website. Eurobudget is an office application written in Java, similar to Microsoft Money or Quicken.

Table 1 shows the statistics of the three target programs. The first program is a piece of commercial software. The second program is a core part of a standard library. The third one is selected from real office software with hundreds of classes and more than 100,000 lines of code in total. All of them are open source. The sizes of these programs are in line with the sizes of target programs used in typical software testing researches such as [20], in which it uses the Siemens suites.

Faulty Versions of Target Programs: To investigate the relative effectiveness of metamorphic testing and assertion checking, we used mutation operators [42] to seed faults to programs. A previous study [20] showed that a set of well-defined mutation operators can simulate the real environment for testing experiments.

Table 1 Statistics of Target Programs

Program	Number of LOC	Number of Methods	Number of Output Affecting Methods
Boyer	241	16	9
BooleanExpression	231	15	12
TxnTableSorter	281	18	15

In our experiment, mutants were seeded using the tool muJava [43]. The tool supports two levels of mutation operators: class level and method level. Class level mutation operators are operators specific to generating faults in object-oriented programs at the class level. Method level mutation operators defined in [26] are operators specific for statement faults. We only seeded method level mutation operators to the programs under study because our experiment focused on unit testing and because this set of operators had been studied extensively in the software engineering research community [5][20][26][28][32][44]. Table 2 list all the mutation operators used in the controlled experiment.

Table 2 Categories of Mutation Operators

Category	Description
AOD	Delete Arithmetic Operator
AOI	Insert Arithmetic Operator
AOR	Replace Arithmetic Operator
ROR	Replace Relational Operator
COR	Replace Conditional Operators
COI	Insert Conditional Operator
COD	Delete Conditional Operator
SOR	Replace Shift Operator
LOR	Replace Logical Operator
LOI	Insert Logical Operator
LOD	Delete Logical Operator
ASR	Replace Assignment Operators

General speaking, muJava examines each statement in a given program and then applies each applicable mutation operator to generate a variant of the program. In other words, for each statement and each applicable mutation operator, it produces a *single-fault* version of the given program. It has been well-recognized in the software engineering research community that single-fault mutants couple well with high-order mutants and real faults and using them to conduct test experiment can adequately simulate realism [20][26]. On the other hand, research on finding an adequate subset of mutation operators to replace the entire set is still going on [44]. Many software engineering researchers continue to use the full set of mutants constructed from a tool to conduct test experiments.

A total of 151 mutants were generated by muJava for the class Boyer, 145 for the class BooleanExpression, and 378 for TxnTableSorter. Note that faults were only seeded into the methods supposedly covered by the test cases for unit testing. Table 3 lists the number of mutants under each category of operators. We created a faulty version for each mutant. Finally, we used all the 674 (151+145+378) single-fault versions in the controlled experiment.

Table 3 Number of Single-Fault Programs by Mutation Operator Category

Program	AOD	AOI	AOR	COD	LOI	ROR	LOR	COR	COI	ASR	Total
Boyer	1	85	14	0	24	16	3	2	1	5	151
BooleanExpression	3	86	3	1	22	27	0	3	0	0	145
TxnTableSorter	8	226	16	0	71	43	2	7	5	0	378

4.3 Experimental Procedure

Before the experiment, the subjects were given a six-hour training to use MT and assertion checking. We carefully monitored the time durations so that the time allocated to train either technique was roughly equal to each other. (We could not have identical durations for both techniques; otherwise, the same testing background such as the concept of test oracles in general would needlessly be introduced twice to the subjects.) The target programs and the tasks to be performed were also presented to the subjects. The subjects were briefed about the main functionality of each target program and the algorithm used, thus simulating the process in real-life in which a tester acquires the background knowledge of the program under test. They were blind to the use of any mutants in the controlled experiment. For each program, the subjects were required to apply MT strictly following the procedure described in Section 3.3, as well as to add assertions to the source code for checking. We did not restrict the number of metamorphic relations and assertions to be associated to individual target programs. The subjects were told to develop metamorphic relations and assertions as they considered suitable, with a view to thoroughly test each target program.

We did not mandate the use of a particular testing case generation strategy, such as all-def-use criterion or random testing or specification-based approach, for either MT or assertion checking. The subjects were simply asked to provide adequate test cases for testing the target programs. This avoided the possibility that some particular test case selection strategy, when applied in large scale, might favor either MT or assertion checking.

We asked the students to submit metamorphic relations, functions to generate follow-up test cases, functions to verify metamorphic relations, test cases for metamorphic testing, source code with inserted assertions, and test cases for assertion checking. They were also asked to report the time costs in applying metamorphic testing and assertion checking. Before testing the faulty versions with these functions, assertions, and test cases, we checked their submissions carefully to ensure that there was no implementation error.

4.4 Threats to Validity

We describe the threats to validity in this section before we present our main results in the next section.

Internal Validity: Internal validity refers to whether the observed effects depend only on the intended experimental variables. For this experiment, we provided the subjects with all the background materials and confirmed with them that they had sufficient time to perform all the tasks. On the other hand, we appreciate that students might be interrupted by minor Internet activities when they performed their tasks. Hence, the time costs reported by the subjects should be viewed and analyzed conservatively. Furthermore, the subjects did not know the nature and details of the faults seeded. This measure ensured that their "designed" metamorphic relations and assertions were unbiased with respect to the seeded faults.

We use test cases provided by our subjects to conduct the experiment. We do not know whether these test cases may favor assertion checking, metamorphic testing, or neither of them. We do not disclose the purpose of the experiment to any subjects, and only request them to produce test cases that they consider sufficient for both metamorphic testing and assertion checking. To address the threat to internal validity, we use all test cases from different subjects on every applicable MR. Since subjects do not communicate with one another in the experiment, this setting helps disassociate test cases from particular MRs.

Readers may be concerned whether the target programs can be faulty. We have carefully checked the classes before the experiment. Furthermore, none of the subjects has reported any errors in the target programs. Another concern is whether the developed MRs may contain faults. To address this threat, we have run all test cases by all subjects as well as our own test cases on all these MRs for the target programs. We observe no failure in the verification exercise. To further address this risk, we have also conducted a verification experiment to explicitly test the mutants of the implementations of the metamorphic relations.

External Validity: External validity is the degree to which the results can be generalized to the testing of real-world systems. The programs used in our experiment are from real-life applications. For example, Eurobudget is widely used and has been downloaded more than 10,000 times from SourceForge. On the other hand, some real-world programs can be much larger and less well documented than the open-source programs studied. More future studies may be in order for the testing of large complex systems using the MT method. We use the MR implementations produced by our subjects. Other testers of other target programs may produce other MR implementations. Additional experiments should always be helpful in improving the generalization of the results that we obtain and present in this paper.

We use Java programs in the experiments, and all MR implementations are naturally written in Java. Although Java programs are widely used in practice, an MR is inherently a property. It may also be intuitive to implement an MR using a rule-based approach via logic programming. It is not immediately obvious to us whether the use of a rule-based approach may produce different comparison results.

We use the test cases produced by the subjects. The use of other schemes (such as statement coverage) may produce different sets of test cases.

Construct Validity: Construct validity refers to whether we are measuring what we intent to measure. We measure the effectiveness of metamorphic testing and assertion checking via a mutation detection ratio. Mutation analysis has been used and validated to be reliable for testing experiments that stimulate real fault scenarios for deterministic, procedural programs (written in C) [20]. The use of mutation detection ratio can be regarded as a reliable measure of the fault detection capability of a testing technique.

In our experiment, to compare metamorphic testing and assertion checking, we use the same test pool and only use the method level of mutation operators to produce mutants in procedural program style. Moreover, the target programs are deterministic; and thus, they produce the same output every time that a program executes a particular test case. Therefore, the failures shown in the outputs are also deterministic. However, our target programs are in Java, which is not the same as the C language. The set of mutation operators is not identical to that used by Andrews et al. [20]. On the other hand, many testing experiments use mutation analysis as the means to assure the effectiveness of various testing techniques.

To measure the time cost for applying MT and assertion checking, we use the time spent by individual subjects on individual target programs. We do not control how a subject conducts their tasks. Thus, a subject may make a mistake when doing a task, find out a similar mistake when working on another task, and then go back to the former task to rectify the first mistake. Thus, a preceding task may be over-estimated in terms of the time spent, while the later task may benefit from the development experience of the preceding task and be under-estimated. We treat this factor as random noise in the experiment. We measure the times reported by each subject on applying MT and on applying assertion checking.

5 Experimental Results

This section presents the experimental results of applying metamorphic testing and assertion checking. They are structured according to the dependent variables presented in the last section.

5.1 Feasibility of MR Development and Assertion Development

A critical and difficult step in applying MT and assertion checking is to develop metamorphic relations and assertions for the target programs. Table 4 reports on the number of metamorphic relations and assertions identified by the subjects for the three target programs. The mean numbers of metamorphic relations developed by the subjects for the respective programs were 2.79, 2.68, and 5.00. The total numbers of distinct metamorphic relations identified by all subjects for the respective programs were 18, 39, and 25. The mean numbers of assertions for the respective programs were 6.96, 11.35, and 10.97.

Table 4 Number of Metamorphic Relations and Assertions

Program	Total	No. of Metamorphic Relations				No. of Assertions			
		Mean	Max	Min	StdDev	Mean	Max	Min	StdDev
Boyer	18	2.79	5	1	1.66	6.96	43	1	8.94
BooleanExpression	39	5.00	12	1	3.01	11.35	49	1	9.69
TxnTableSorter	25	2.68	7	1	1.59	10.97	36	2	10.97

First, we observe that all the subjects could properly create metamorphic relations and assertions after training. We further inspect their metamorphic relations and assertions, and find that many of the identified artifacts overlap among subjects. Take Boyer as an example. There are 38 subjects in total. They collectively identify 18 distinct metamorphic relations, and on average, each subject identifies 2.79 metamorphic relations. In other words, if all the metamorphic relations identified were distinct, there should be 108 metamorphic relations. It means that, on average, each distinct metamorphic relation is discovered by six subjects (or 15.7% of the population). We also observe a similar result for assertion checking. This result is encouraging. It indicates that the identification of metamorphic relations can be practical and may share among different developers. It further answers another important research question on whether the same metamorphic relation can be discovered by more than one subject. The answer is “yes”.

To observe the variations in the feasibility of discovering metamorphic relations and assertions, we further normalize the standard derivations against the corresponding mean values in Table 4 for each of the programs. The results are shown in Table 5. We observe that the standard deviations for discovering metamorphic relations are much larger than those for discovering assertions. In addition, we observe that the normalized standard deviations for discovering metamorphic relations across the three programs are quite consistent (close to 0.60 in each case). On the other hand, for assertions, the standard deviations trends vary from 0.20 to 0.30, which indicate a relatively larger fluctuation among programs. This initial finding may indicate that discovering metamorphic relations can be less dependent on the type of program being studied than discovering assertions. In other words, it suggests that there may be some hidden dominant factors (independent of the nature of target programs) governing the discovery of metamorphic relations. It will be interesting to identify these factors in the future.

On the other hand, we observe from Table 5 that the absolute values of the normalized standard deviations for discovering assertions are much smaller than those of metamorphic relations. It shows that our subjects produce more predictable number of assertions. It may give project managers good guidelines to allocate project resources if they assign their programmers to do assertion checking in their software applications.

Table 5 Normalized Standard Derivations

Program	Metamorphic Relation	Assertion Checking
Boyer	0.59	0.21
BooleanExpression	0.60	0.20
TxnTableSorter	0.59	0.30

5.2 Size and Granularity of Metamorphic Relations and Assertions per Program

In general, the subjects could identify a larger number of assertions than metamorphic relations. As shown in Table 4, the maximum number of metamorphic relations discovered by subjects is almost the same as the mean number of assertions discovered by subjects. This suffices to indicate that there is a significant difference between the numbers of artifacts produced by the two testing methods.

We also observe that the subjects' abilities to identify metamorphic relations and assertions vary. This is understandable and agrees with the intuition that different developers may have quite diverse programming abilities. Take BooleanExpression as an example. Some subjects can identify 12 metamorphic relations and 49 assertions, while some others can only identify one metamorphic relation and one assertion.

We further observe from Table 4 that, for the three target programs, the ratios of the mean number of identified metamorphic relations to the mean number of identified assertions are 0.40, 0.44, and 0.24, respectively. If the effectiveness between the use of metamorphic testing and the use of assertion checking to identify failures is comparable, these ratios indicate that metamorphic relations can achieve a more coarse-grained granularity than assertions. If so, we believe that MT helps developers raise the level of abstraction more than assertion checking does. Our data analysis to be presented in the next section will validate whether the effectiveness of the two methods are comparable.

5.3 Comparison on Fault Detection Capabilities

We use the subjects' metamorphic relations, assertions, and source and follow-up test cases to test the faulty versions of the target programs. The mutation detection ratio [20][26][42] is used to compare the fault detection capabilities of MT and assertion checking strategies. The *mutation detection ratio* of a test set is defined as the number of mutants detected by the test set over the total number of mutants [42]. For metamorphic testing, a mutant is detected if a source test case and follow up test cases executed on the mutant do not satisfy some metamorphic relations. For assertion checking, a mutant is detected if a mutated statement is executed by a test case to enter an erroneous state that triggers an assertion statement.

For the sake of fairness, we applied these two methods to the *same* set of test cases separately. The source and follow-up test cases from metamorphic testing were both applied to assertion checking.

The average sizes of the test suites (including source and follow-up test cases) used by all students for the three programs were 19.9, 22.2, and 16.8, respectively. We also analyzed all the mutants manually before testing and removed the equivalent mutants. There were 19, 18, and 61 equivalent mutants for program Boyer, BooleanExpression, and TxnTableSorter, respectively. We did not include them when calculating mutation detection ratios as these mutants cannot be detected by any test cases.

Table 6 Mutation Detection Ratios for Metamorphic Testing and Assertion Checking

Program	Metamorphic Testing					Assertion Checking					Result of p-value of Mann-Whitney Test
	Mean	Max	Min	StdDev	Aggregate	Mean	Max	Min	StdDev	Aggregate	
Boyer	60%	93%	44%	0.13	98%	40%	66%	27%	0.12	81%	< 0.001
BooleanExpression	63%	89%	46%	0.11	95%	39%	66%	30%	0.10	78%	< 0.001
TxnTableSorter	59%	74%	32%	0.14	83%	37%	58%	22%	0.11	63%	< 0.001

Table 6 reports on the mutation detection ratios for each program using the two testing methods. It shows that the mutation detection ratios by applying MT ranged from 44% to 93% for program Boyer, from 46% to 89% for program BooleanExpression, and from 32% to 74% for program TxnTableSorter.

Under the “Aggregate” columns are the percentages of mutants detected by all subjects. For MT, the mutation detection ratios were 98%, 95%, and 83%, respectively. Each entry was *significantly* better than the corresponding mutation detection ratio for assertion checking. This result, again, is encouraging.

The p-value of the standard Mann-Whitney test was less than 0.001 in all cases. Hence, we reject the null hypothesis H_0 on the effectiveness of fault detection at a 99.9% confidence level. In other words, MT may not only be comparable to assertion checking, but outperforms the latter. We have used the same set of test cases when applying the Mann-Whitney test.

This setting and hypothesis testing result indicate that the difference is attributed by the ability to *violate* the constraints specified via metamorphic relations and those specified via assertion checking. We observe that the difference between the two testing methods in our experiment is whether the constraint is specified for one execution or for multiple executions. The former type of constraint is for assertion checking, and the latter type is for metamorphic relation. In the other words, the result indicates that using the test results of multiple executions to identify failures collectively is more effective than just using one execution.

Although our empirical results show that metamorphic testing can be effective, there is a need to develop systematic methods for creating metamorphic relations and assertions (because individual tester’s results were lower than the aggregated results of all testers in either approach). The average differences between the mean column and the aggregate column for MT and assertion checking were 41.3% and 35.3%, respectively. The standard derivations did not differ much statistically. They ranged from 0.10 to 0.14, as shown in Table 6.

5.4 Comparison of Time Cost

We would like to compare the time costs between metamorphic testing and assertion checking. From the subjects’ submissions, we found that they spent less time on applying assertion checking than metamorphic testing.

Table 7 Statistics of Time Costs for Applying MT and Assertion Checking

		Smallest Observation	Lower Quartile	Lower Notch	Median	Upper Notch	Upper Quartile	Largest Observation
Boyer	**MT**	0.58	1.73	1.99	2.51	5.01	5.11	9.82
	Assertion	0.58	1.03	1.03	1.48	1.99	2.12	2.18
BooleanExpression	**MT**	0.32	2.25	2.51	3.28	6.03	8.02	12.71
	Assertion	0.45	1.35	1.48	1.99	3.02	5.01	7.77
TxnTableSorter	**MT**	0.26	2.51	3.02	3.98	6.03	6.99	11.68
	Assertion	0.52	1.03	1.28	1.99	3.02	3.98	6.74

Table 7 shows the statistics of the time costs for applying the respective strategies to the target programs. Each entry in the column “Smallest Observation” stands for the smallest value (time cost in terms of hours) in the respective data set. Each entry in the column “Largest Observation” stands for the largest value in the respective data set. Each entry under “Median” captures the 50th percentile in the data. The entries under “Lower Quartile” and “Upper Quartile” capture the values of the 25th and 75th percentiles (in the order from small to large) in the data, respectively. The entries under “Lower Notch” and “Upper Notch” display the variability of the median in the data set.

We observe from Table 7 several interesting tradeoffs between MT and assertion checking. First, the smallest observation in assertion checking is consistently larger than that in MT. Although applying MT is apparently more complex than assertion checking, this result shows that, for the most effective testers, the effort to design and implement metamorphic relations is less than the effort to design and implement assertions. Second, for each of the three target programs, the median and the largest observation in MT are always greater than the corresponding values in assertion checking. It indicates that designing and implementing MRs is generally more time-consuming than designing and implementing assertions. Third, from the lower quartiles and upper quartiles in MT and assertion checking for these programs, we further observe that the time spent on MT varies more drastically than the time on assertion checking.

Intuitively, many developers have developed skills to understand program logic from source code and are comfortable in conducting program comprehension. Furthermore, developers are used to modifying an existing program to implement new changes to the source code. In view of the above intuition, we believe that adding assertions to source code is a more familiar and handy task for the subjects than formulating and implementing MRs.

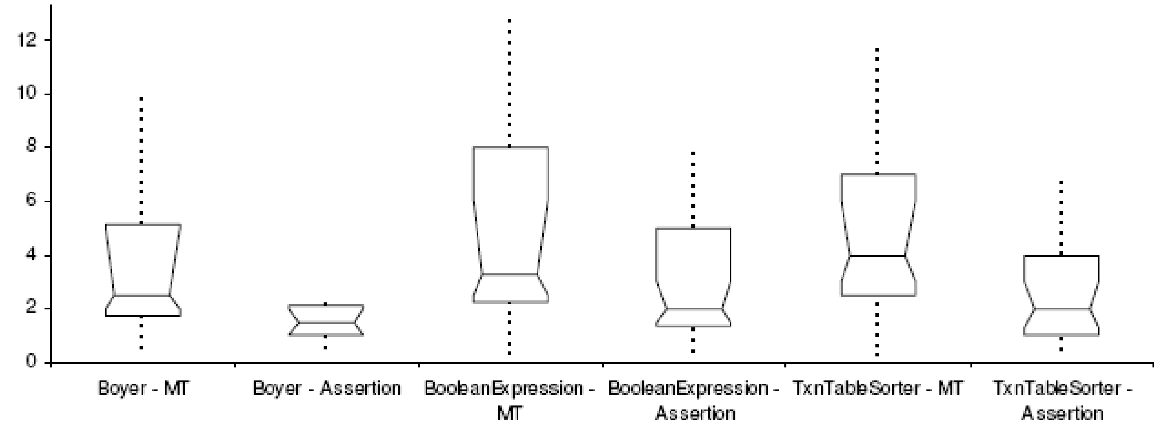

Figure 2 Box-and-Whisker Plots of Time Costs for Applying MT and Assertion Checking

To analyze the differences between these two testing approaches to alleviate the test oracle problem, we further represent their time costs using box-and-whisker plots. Figure 2 shows the plots for applying the respective strategies to the target programs. The time cost for MT includes the time to identify and formulate test cases, write functions to generate follow-up test cases, and write functions to verify the identified metamorphic relations. The time cost for assertion checking includes the time spent on adding assertions to the source code.

The vertical axis of Figure 2 shows the time cost in number of hours. The bottom and top horizontal lines of each box indicate the lower and upper quartiles. The whiskers, drawn as dotted vertical lines, show the full range of the data. The median is drawn as a horizontal line inside each box. A notch is added to each box to show the uncertainty interval for each median. If two median notches do not overlap, it indicates that there is a statistically significant difference between the two medians at a 95% confidence level.

For Boyer and TxnTableSorter, there is a significantly difference between the times spent in applying metamorphic testing and assertion checking. The difference is less statistically significant for BooleanExpression. The exact values of the respective notches can be found in Table 7.

The difference in time cost is acceptable for a number of reasons. First, the time costs for MT implementations include the generation of follow-up test cases, whereas the time costs for assertion checking do not include the generation of any test cases. Second, some subjects have had prior experience in assertion checking. We believe that the extra time spent on developing programs to generate follow-up test cases have paid off because, as discussed in Section 5.2, these (follow-up) test cases have demonstrated to be very useful in detecting failures of the target programs. Furthermore, although there is a statistically significant difference in time costs (especially if we view Table 7 in relative terms), we also note that the actual median difference in absolute terms range between one to two hours in the experiment.

Figure 2 further indicates that the time cost for applying MT to object-oriented testing at the class level is acceptable compared to that of assertion checking. When we consolidate the comparisons in Sections 5.2 and 5.3, we find that MT provides a stronger oracle check with a tradeoff of slightly more time for preparation.

5.5 Comparison of MT with and without Faulty MR Implementations

As we have highlighted in Section 3.1, an MR is a property that the correct version of a program under test should exhibit. To apply MT automatically, testers need to execute the implementations of the MRs for the program under test. In the controlled experiment, these MR implementations are constructed by the subjects. It is crucial to know whether MT can still be effective if MR implementations can be faulty.

We thus conducted a follow-up experiment to validate whether MT is robust enough if faulty metamorphic relations are used to detect failures in the subject programs. We used the set of mutation operators of muJava mentioned above to generate single-fault mutants of the MR implementations. In total, muJava produced 88, 71, and 89 MR mutants for the three subject programs, respectively. If an MR mutant cannot be killed by any test case, we excluded such a mutant from the follow-up experiment. We also excluded similar target program mutants. We then selected a test suite of 20 test cases randomly from the test pool for each target program and computed the mutation detection ratio accordingly. We note that, in this validation experiment, a revealed failure may be a mistake (namely, a false positive case) produced by a faulty MR implementation, a failure of the faulty target program, or both. We repeated the experiment by selecting the test suites 10 times.

Table 8 Mutation Detection Ratios for Metamorphic Testing with and without Faulty Metamorphic Relation Implementations

Program	With Correct MR Implementations Only					With Both Correct and Faulty MR Implementations				
	Mean	**Max**	**Min**	**StdDev**	**Median**	**Mean**	**Max**	**Min**	**StdDev**	**Median**
Boyer	59%	100%	2%	0.25	56%	95%	100%	85%	0.03	95%
BooleanExpression	72%	100%	34%	0.27	85%	91%	100%	80%	0.08	94%
TxnTableSorter	66%	100%	6%	0.22	57%	91%	100%	67%	0.05	90%

The result is shown in Table 8. First, if we only use correct MRs to identify failures, the mean fault detection rate in the validation experiment is close to the mutation detection rate shown in Table 6. It indicates that the results of the validation experiment are comparable to the above-mentioned experiment that compares MT and assertion checking. Second, if MR implementations can be faulty, the mean value is much higher (consistently over 90% as shown in the rightmost column of Table 8). The result indicates that a test suite is likely to detect problems in the combination of a faulty target program and a set of faulty MRs. This finding is encouraging because MT can still be reasonably applied even if some MR implementations may be faulty. If a faulty MR implementation can be debugged successfully, we believe that the failure detection rate of the test suite will drop, as indicated by the comparison in Table 7. However, fixing the faults in the MR implementations will incur additional time cost. It may make the difference in time cost between metamorphic testing and assertion checking more noticeable. Thus, it warrants more study to find the extent that testers should stop further maintenance of a faulty MR implementation in order to balance the development cost and product quality.

5.6 Further Discussions on MT

In general, we observe that the more MRs being used, the higher will be the mutation detection ratio. As we have indicated in Section 5.2, there is a need to propose more systematic methods to construct the implementation of metamorphic relations. The utilization of an MR implementation also increases as testers increase the number of initial test cases applicable to the MR. Since the resources in software testing are often limited, it is also worth investigating the number of test cases adequate for MT.

Moreover, testers may apply a number of metamorphic relations in order to test a program. In general, different metamorphic relations have non-identical fault detection capabilities. Let us, for instance, analyze the experimental results of the Boyer program. The subjects have identified 18 metamorphic relations in total. We observe that four subjects have only identified one and the same metamorphic relation (MR_1 in Table 9). The implementation of this metamorphic relation constructs a follow-up test case by appending an arbitrary string to the string in the initial test case and reusing the given pattern in the initial test case. It also checks whether the Boyer program over the two test cases will give the same outputs if the program locates successfully the given pattern in the initial string. The mutation detection ratios resulting from these MR implementations by the subjects are no more than 60% no matter how many test cases they used. We also find that some subjects using the other metamorphic relations (MR_2 and MR_3 in Table 9) achieve mutation detection ratios higher than 80%, although they only propose four initial test cases. It indicates that the quality of metamorphic relations can be a key factor in determining the effectiveness of MT.

Table 9 Examples of Metamorphic Relations for Program Boyer

Index	Metamorphic Relation
MR_1	*If* $(x_1 = concatenate\ (x_2, x_3)) \wedge (find\ (x_2, x_4) > -1)$, *then* $find\ (x_1, x_4) = find\ (x_2, x_4)$.
MR_2	*If* $(x_1 = concatenate\ (x_2, x_3)) \wedge (find\ (x_2, x_4) = -1) \wedge (find\ (x_3, x_4) > -1)$, *then* $find\ (x_1, x_4) \leq length\ (x_2) + find\ (x_3, x_4)$.
MR_3	*If* $(x_1 = concatenate\ (x_2, x_3)) \wedge (find\ (x_1, x_4) = length\ (x_2))$, *then* $find\ (x_3, x_4) = 0$.

The function *concatenate* (x, y) returns the result of concatenating string x and string y. The function *find* (x, y) returns the zero-based index of string y within the string x if x contains y; otherwise, it returns -1.

6 Conclusion

This paper has reported a controlled experiment to study the application of metamorphic testing (MT) and assertion checking as the means to alleviate the test case problem. A main objective is to evaluate whether MT is a useful and viable strategy and to assess its cost and effectiveness. We choose to compare MT with a popular testing method, namely, assertion checking. The experiment indicates that, after training, the subjects could apply MT to test programs effectively. For all the three open-source programs under study, the subjects could identify many useful metamorphic relations. The results also suggest that MT is a more effective testing strategy than assertion checking in terms of fault detection capability. The time cost of applying MT is acceptable when compared with assertion checking. However, assertion checking is more efficient. Our study also reveals that the granularity of MRs is coarser than that of assertion checking. It may indicate that MRs provide a high level of abstraction for testers to deal with testing tasks.

Future research includes the following: In our experiments, the subjects must identify metamorphic relations and develops programs manually to apply MT. It will be desirable to automate, even in part, the formulation and generation of metamorphic relations. Other future experiments include the impact of experience levels of subjects on applying MT, and the use of other subjects, programs and MRs. It is also interesting to study how MT integrates with test case adequacy criteria, and the role of program development environments to lower the barrier to applying MT in practice. Our empirical study only examines the effectiveness and time cost of metamorphic testing and assertion checking. We have not formulated the underpinning theory to explain the differences. We hope that our study provides an initial set of empirical evidence for researchers to explore this inadequately researched and yet important area.

Acknowledgement

We would like to thank Fan Liang of The University of Hong Kong for conducting the validation experiment.

References

[1] 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*) in conjunction with the *14th ACM SIGSOFT Symposium on Foundations of Software Engineering* (*SIGSOFT 2006/FSE-14*), pages 6–13. ACM Press, New York, NY, 2006.

[2] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold, New York, NY, 1990.

[3] E. J. Weyuker. On testing non-testable programs. *The Computer Journal*, 25 (4): 465–470, 1982.

[4] T. Y. Chen, J. Feng, and T. H. Tse. Metamorphic testing of programs on partial differential equations: a case study. In *Proceedings of the 26th Annual International Computer Software and Applications Conference* (*COMPSAC 2002*), pages 327–333. IEEE Computer Society Press, Los Alamitos, CA, 2002.

[5] W. K. Chan, S. C. Cheung, J. C. F. Ho, and T. H. Tse. PAT: a pattern classification approach to automatic reference oracles for the testing of mesh simplification programs. *Journal of Systems and Software*, 2008. doi.10.1016/j.jss.2008.07.019.

[6] W. K. Chan, J. C. F. Ho, and T. H. Tse. Piping classification to metamorphic testing: an empirical study towards better effectiveness for the identification of failures in mesh simplification programs. In *Proceedings of the 31st Annual International Computer Software and Applications Conference* (*COMPSAC 2007*), volume 1, pages 397–404. IEEE Computer Society Press, Los Alamitos, CA, 2007.

[7] T. H. Tse, S. S. Yau, W. K. Chan, H. Lu, and T. Y. Chen. Testing context-sensitive middleware-based software applications. In *Proceedings of the 28th Annual International Computer Software and Applications Conference* (*COMPSAC 2004*), volume 1, pages 458–465. IEEE Computer Society Press, Los Alamitos, CA, 2004.

[8] H. Lu, W. K. Chan, and T. H. Tse. Testing context-aware middleware-centric programs: a data flow approach and an RFID-based experimentation. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2006/FSE-14*), pages 242–252. ACM Press, New York, NY, 2006.

[9] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. A metamorphic approach to integration testing of context-sensitive middleware-based applications. In *Proceedings of the 5th International Conference on Quality Software* (*QSIC 2005*), pages 241–249. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[10] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. Integration testing of context-sensitive middleware-based applications: a metamorphic approach. *International Journal of Software Engineering and Knowledge Engineering*, 16 (5): 677–703, 2006.

[11] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4 (2): 60–80, 2007.

[12] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Fault-based testing without the need of oracles. *Information and Software Technology*, 45 (1): 1–9, 2003.

[13] R. V. Binder. *Testing Object-Oriented Systems: Models, Patterns, and Tools*. Addison Wesley, Reading, MA, 2000.

[14] R. N. Taylor. Assertions in programming languages. *ACM SIGPLAN Notices*, 15 (1): 105–114, 1980.

[15] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Semi-proving: an integrated method based on global symbolic evaluation and metamorphic testing. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2002*), pages 191–195. ACM Press, New York, NY, 2002.

[16] 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. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[17] Y. Sun and E. L. Jones. Specification-driven automated testing of GUI-based Java programs. In *Proceedings of the 42nd Annual Southeast Regional Conference* (*ACM-SE 42*), pages 140–145. ACM Press, New York, NY, 2004.

[18] W. K. Chan, S. C. Cheung, and K. R. P. H. Leung. Towards a metamorphic testing methodology for service-oriented software applications. The 1st International Conference on Services Engineering (SEIW 2005). In *Proceedings of the 5th International Conference on Quality Software* (*QSIC 2005*), pages 470–476. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[19] R. L. Cobleigh, G. S. Avrunin, and L. A. Clarke. User guidance for creating precise and accessible property specifications. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2006/FSE-14*), pages 208–218. ACM Press, New York, NY, 2006.

[20] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments?. In *Proceedings of the 27th International Conference on Software Engineering* (*ICSE 2005*), pages 402–411. ACM Press, New York, NY, 2005.

[21] D. Chapman. A program testing assistant. *Communications of the ACM*, 25 (9): 625–634, 1982.

[22] S. Ar, M. Blum, B. Codenotti, and P. Gemmell. Checking approximate computations over the reals. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing* (*STOC ’93*), pages 786–795. ACM Press, New York, NY, 1993.

[23] M. Blum and S. Kannan. Designing programs that check their work. *Journal of the ACM*, 42 (1): 269–291, 1995.

[24] M. Blum, M. Luby, and R. Rubinfeld. Self-testing/correcting with applications to numerical problems. *Journal of Computer and System Sciences*, 47 (3): 549–595, 1993.

[25] Q. Xie and A. M. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology*, 16 (1): Article No. 4, 2007.

[26] A. J. Offutt, A. Lee, G. Rothermel, R. H. Untch, and C. Zapf. An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, 5 (2): 99–118, 1996.

[27] R. Helm, I. M. Holland, and D. Gangopadhyay. Contracts: specifying behavioral compositions in object-oriented systems. In *Proceedings of the 5th Annual Conference on Object-Oriented Programming Systems, Languages, and Applications* (*OOPSLA ’90*), *ACM SIGPLAN Notices*, 25 (10): 169–180, 1990.

[28] L. C. Briand, M. Di Penta, and Y. Labiche. Assessing and improving state-based class testing: a series of experiments. *IEEE*

Transactions on Software Engineering, 30 (11): 770–783, 2004.

[29] B. Meyer. Applying 'design by contract'. *IEEE Computer*, 25 (10): 40–51, 1992.

[30] M. Last, M. Friedman, and A. Kandel. The data mining approach to automated software testing. In *Proceedings of the 9th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (*KDD 2003*), pages 388–396. ACM Press, New York, NY, 2003.

[31] M. Vanmali, M. Last, and A. Kandel. Using a neural network in the software testing process. *International Journal of Intelligent Systems*, 17 (1): 45–62, 2002.

[32] P. Francis, D. Leon, M. Minch, and A. Podgurski. Tree-based methods for classifying software failures. In *Proceedings of the 15th International Symposium on Software Reliability Engineering* (*ISSRE 2004*), pages 451–462. IEEE Computer Society Press, Los Alamitos, CA, 2004.

[33] A. Podgurski, D. Leon, P. Francis, W. Masri, M. Minch, J. Sun, and B. Wang. Automated support for classifying software failure reports. In *Proceedings of the 25th International Conference on Software Engineering* (*ICSE 2003*), pages 465–475. IEEE Computer Society Press, Los Alamitos, CA, 2003.

[34] J. F. Bowring, J. M. Rehg, and M. J. Harrold. Active learning for automatic classification of software behavior. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2004*), pages 195–205. ACM Press, New York, NY, 2004.

[35] W. K. Chan, M. Y. Cheng, S. C. Cheung, and T. H. Tse. Automatic goal-oriented classification of failure behaviors for testing XML-based multimedia software applications: an experimental case study. *Journal of Systems and Software*, 79 (5): 602–612, 2006.

[36] S. Beydeda. Self-metamorphic-testing components. In *Proceedings of the 30th Annual International Computer Software and Applications Conference* (*COMPSAC 2006*), volume 1, pages 265–272, IEEE Computer Society Press, Los Alamitos, CA, 2006.

[37] P. Wu. Iterative metamorphic testing. In *Proceedings of the 29th Annual International Computer Software and Applications Conference* (*COMPSAC 2005*), volume 1, pages 19–24. IEEE Computer Society Press, Los Alamitos, CA, 2005.

[38] C. Murphy. Using runtime testing to detect defects in applications without test oracles. In *Companion to Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2008/FSE-16*)*,* ACM Press, New York, NY, 2008.

[39] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. Application of metamorphic testing in numerical analysis. In *Proceedings of the IASTED International Conference on Software Engineering* (*SE '98*), pages 191–197. ACTA Press, Calgary, Canada, 1998.

[40] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta. A controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Transactions on Software Engineering*, 27 (12): 1134–1144, 2001.

[41] M. Vokáč, W. Tichy, D. I. K. Sjoberg, E. Arisholm, and M. Aldrin. A controlled experiment comparing the maintainability of program designed with and without design patterns: a replication in a real programming environment. *Empirical Software Engineering*, 9 (3):149–195, 2004.

[42] W. E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, SE-8 (4): 371–379, 1982.

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

[44] A. S. Namin, J. H. Andrews, and D. J. Murdoch. Sufficient mutation operators for measuring test effectiveness. In *Proceedings of the 30th International Conference on Software Engineering* (ICSE 2008)*,* pages 351–360, ACM Press, New York, NY, 2008.