An Empirical Comparison between Direct and Indirect Test Result Checking Approaches*†‡§

Peifeng Hu
The University of Hong Kong
Pokfulam
Hong Kong
pfhu@cs.hku.hk

Zhenyu Zhang
The University of Hong Kong
Pokfulam
Hong Kong
zyzhang@cs.hku.hk

W. K. Chan
City University of Hong Kong
Tat Chee Avenue
Hong Kong
wkchan@cs.cityu.edu.hk

T. H. Tse
The University of Hong Kong
Pokfulam
Hong Kong
thtse@cs.hku.hk

ABSTRACT

An oracle in software testing is a mechanism for checking whether the system under test has behaved correctly for any executions. In some situations, oracles are unavailable or too expensive to apply. This is known as the oracle problem. It is crucial to develop techniques to address it, and metamorphic testing (MT) was one of such proposals. This paper conducts a controlled experiment to investigate the cost effectiveness of using MT by 38 testers on three open-source programs. The fault detection capability and time cost of MT are compared with the popular assertion checking method. Our results show that MT is cost-efficient and has potentials for detecting more faults than the assertion checking method.

Categories and Subject Descriptors

D.2.5 [**Software Engineering**]: Testing and Debugging—*Testing tools*; D.2.8 [**Software Engineering**]: Metrics—*Product metrics*

General Terms

Experimentation

Keywords

Metamorphic testing, test oracle, controlled experiment, empirical evaluation

1. INTRODUCTION

Software testing assures programs by executing test cases over the programs with the intent to reveal failures [3]. To do so, software testers need to evaluate test results through an *oracle*, which is a mechanism for checking whether a program has behaved correctly [35]. In many situations, however, oracles are unavailable or too expensive to apply. This is known as the *oracle problem* [35]. Usually, the main purpose of implementing a specific program is to compute unknown results. If the expected results could easily be computed through other automatic means, then there would not be a need to implement the program in the first place. On the other hand, manual checking of program outputs is slow, ineffective, and costly, especially for 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" [27].

As we shall review in Section 2, assertion checking [4] and *metamorphic testing* (MT) [9, 17, 18, 11] are techniques to alleviate the oracle problem. Assertion checking verifies the result or intermediate program states of a test case. It directly confirms the execution behavior of a program in terms of a checking condition. MT takes another direction, which verifies follow-up test cases based on existing test cases. It cross-checks the test results of existing 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. It would be interesting to compare the two approaches on their effectiveness to identify failures.

† This research is supported in part by a grant of the Research Grants Council of Hong Kong (project no. HKU 7145/04E), a grant of City University of Hong Kong and a grant of The University of Hong Kong.

‡ All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2183. Fax: (+852) 2557 8447. Email: thtse@cs.hku.hk.

§ Part of the work was done when Chan was with The Hong Kong University of Science and Technology, Clear Water Bay, Hong Kong.

SOQUA '06, November 6, 2006, Portland, OR, USA.

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. [16] reported on the testing of programs for solving partial differential equations. They further investigated the integration of metamorphic testing with fault-based testing and global symbolic evaluation [18]. Gotlieb and Botella [22] developed an automated framework to check against a restricted class of metamorphic relations. Tse and others applied metamorphic approach to the unit testing [33] and integration testing [10] of context-sensitive middleware-based applications. Chan et al. [13, 14] developed a metamorphic approach to the online testing of service-oriented software applications. Throughout these studies, both the testing and the evaluation of experimental results were conducted by the researchers themselves. The programs under test were from academic sources and relatively small. There is a need for systematic empirical research on how well MT can be applied in practical situations and how effective it is compared with other testing strategies [1].

Like other comparisons of testing strategies such as between control flow and data flow criteria [21] and among different data flow criteria [25], 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 effort for applying MT?

This paper reports and discusses the results in such a controlled experiment. We restricted the scope to object-oriented testing at the class level [4]. The subjects were 38 postgraduate students enrolled in an advanced software testing course. Before doing the experiment, they were taught the concepts of MT and a reference strategy — assertion checking [4] — 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 faulty versions of the target programs to assess the capability of these two testing strategies in detecting faults [1]. Results were analyzed to compare the costs and effectiveness between MT and assertion checking.

The main contributions of this paper are four-fold: (i) It is the first controlled experiment to study the above questions. (ii) The experiment shows that metamorphic testing is more effective than assertion checking for identifying faults for object-oriented programs. (iii) It confirms the belief that the subjects can formulate metamorphic relations and implement MT without much difficulty. In fact, the experiment shows that all subjects manage to propose metamorphic relations after a brief introduction, and identical or very similar metamorphic relations are proposed by different subjects. (iv) It also indicates that metamorphic testing is worth applying in terms of time cost.

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

[1] Other researchers have evaluated the selection of metamorphic relations. However, their work is not yet publicly accessible at the time of submission of this paper. Thus, we shall exclude them from our discussions.

2. RELATED WORK

Many approaches have been proposed to alleviate the test oracle problem. Instead of checking the output directly, these approaches generated various types of oracle to verify the correctness of a program. Chapman [15] suggested that a previous version of a program could be used to verify the correctness of the current version. Weyuker [35] suggested checking whether some identity relations would be preserved by the program under test. Blum and others [6, 2] 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 [5]. Xie and Memon [36] studied different types of oracle for graphic user interface (GUI) testing. Binder [4] discussed four categories and eighteen oracle patterns in object-oriented program testing.

Assertion checking [32] is another method to verify the execution results of programs. An assertion, which is embedded directly in the source code, 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. Assertion checking has been widely used in object-oriented testing. For example, state invariants [4, 23], represented by assertions, can be used to check the stated-based behaviors of a system. Briand et al. [8] 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.

Some researchers have proposed to prepare test specifications, either manually or automatically, to alleviate the test oracle problem. Memon et al. [28] assumed that a test specification of internal object interactions was available and used it to identify non-conformance of the execution traces. This type of approach is common in conformance testing for telecommunication protocols. Sun et al. [31] proposed a similar approach to test the harnesses of applications. Last and others [24, 34] trained pattern classifiers to learn the casual input-output relationships of a legacy system. They then used the classifiers as test oracles. Podgurski and others [30, 20] classified failure reports into categories via classifiers, and then refined the classification by further means. Bowring et al. [7] used a progressive approach to train a classifier to help regression testing. Chan et al. [12] used classifiers to identify different types of behaviors related to the synchronization failures of objects in a multimedia application.

3. PRELIMINARIES OF METAMORPHIC RELATIONS AND TESTING

This section introduces metamorphic testing. As we have briefed in Section 1, metamorphic testing relies on a checking condition that relates multiple test cases and their results in order to verify whether any failures are revealed. Such a checking condition is known as a *metamorphic relation*. We shall first revisit metamorphic relations and then discuss how they are used in the metamorphic approach to software testing.

3.1 Metamorphic Relations

A metamorphic relation (MR) is an existing or expected relation over a set of distinct inputs and their corresponding outputs for multiple executions of the target function [17]. Consider, for

instance, the sine function. For any inputs x_1 and x_2 such that $x_1 + x_2 = \pi$, we must have $\sin x_1 = \sin x_2$.

Definition 1 (Metamorphic Relation) [11] *Let $\langle x_1, x_2, \ldots, x_k \rangle$ be a series of inputs to a function f, where $k \geq 1$, and $\langle f(x_1), f(x_2), \ldots, f(x_k) \rangle$ be the corresponding series of results. Suppose $\langle f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_m}) \rangle$ is 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 $\langle f(x_{k+1}), f(x_{k+2}), \ldots, f(x_n) \rangle$ be the corresponding series of results. Suppose, further, that there exists relations $r(x_1, x_2, \ldots, x_k, f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_m}), x_{k+1}, x_{k+2}, \ldots, x_n)$ 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. We say that*

$$\begin{aligned} \textbf{MR} = \{ \quad & (x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)) \mid \\ & r(x_1, x_2, \ldots, x_k, f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_m}), \\ & \quad x_{k+1}, x_{k+2}, \ldots, x_n) \\ & \rightarrow r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)) \} \end{aligned}$$

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

$$\begin{aligned} \textbf{MR}: \quad & \textit{If } r(x_1, x_2, \ldots, x_k, f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_m}), \\ & \quad x_{k+1}, x_{k+2}, \ldots, x_n) \\ & \textit{then } r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)). \end{aligned}$$

Furthermore, x_1, x_2, …, x_k are known as source test cases and x_{k+1}, x_{k+2}, …, 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 so on.

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. Is there a systematic methodology guiding testers to formulate metamorphic relations like the methodologies that guide systems analysts to specify requirements? This remains an open 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 are "successful test cases", which do not reveal any failure. These successful test cases have been considered useless in conventional testing. Metamorphic testing (MT) uses information from such successful test cases, which will be referred to as *source test cases*.

Consider a program p for a target function f in the input domain D. A set of source test cases $T = \{t_1, t_2, \ldots, t_k\}$ can be selected according to any test case selection strategy. Executing the program p on T produces outputs $p(t_1), p(t_2), \ldots, p(t_k)$. When there is an oracle, the test results can be verified against $f(t_1), f(t_2), \ldots, f(t_k)$. If these results reveal any failure, testing stops. On the other hand, when there is no oracle or when no failure is revealed, the metamorphic testing approach can continue to be applied to automatically generate *follow-up test cases* $T' = \{t_{k+1}, t_{k+2}, \ldots, t_n\}$ based on source test cases T, so that the program can be verified against metamorphic relations. For example, given a source test case x_1 for a program that implements the sine function, we can construct a follow-up test case x_2 based on the metamorphic relation $x_1 + x_2 = \pi$.

Definition 2 (Metamorphic Testing) [11] *Let P be an implementation of a target function f. The* metamorphic testing *of the metamorphic relation*

$$\begin{aligned} \textbf{MR}: \quad & \textit{If } r(x_1, x_2, \ldots, x_k, f(x_{i_1}), f(x_{i_2}), \ldots, f(x_{i_m}), \\ & \quad x_{k+1}, x_{k+2}, \ldots, x_n), \\ & \textit{then } r'(x_1, x_2, \ldots, x_n, f(x_1), f(x_2), \ldots, f(x_n)) \end{aligned}$$

involves the following steps: (1) *Given a series of source 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 $r(x_1, x_2, \ldots, x_k, P(x_{i_1}), P(x_{i_2}), \ldots, P(x_{i_m}), x_{k+1}, x_{k+2}, \ldots, x_n)$ over the implementation P.* (2) *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 [22] developed an automated framework for a subclass of metamorphic relations. The framework translates a specification into a constraint logic programming language program. Test cases can be automatically be generated according to metamorphic testing. Their framework only works on a restricted subset of the C language and is not applicable to test cases involving objects. Since we want to apply MT to test real-world object-oriented programs, we adopt the original procedure [9] as follows:

Firstly, 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 source 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 source 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 firstly 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?

(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?

4.2 Experimental Design

Our experiment identifies four independent and three dependent variables. The independent variables are *testing strategies*, *subjects*, *target programs*, and *faulty versions of target programs*. The dependent variables are *effort in terms of 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 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.

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. It showed that most of them had real-life or academic experience in object-oriented design, Java programming, software testing, and assertion checking.

Figure 1 lists the survey result. As most of subjects were knowledgeable about object-oriented design and Java programming, they were deemed to be competent in the experimental tasks. 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 [2]. 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 value. For example, the evaluation result of "!(true && false) || true" is "true".

[2] URL http://mindprod.com/products1.html.

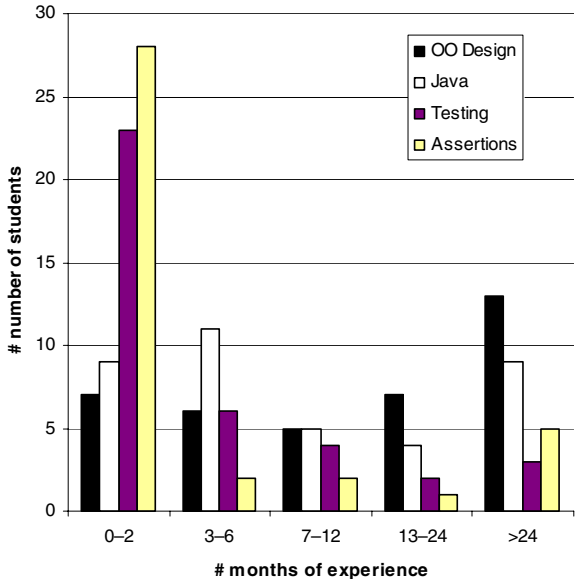

Figure 1: Experience of Subjects in Object-Oriented Design, Java, Testing, and Assertions

Table 1: Statistics of Target Programs

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

The program is part of a popular open-source project jboolexpr [3] in SourceForge [4], which is the largest open-source project website. The target program is a core part of the project.

The third target program is TxnTableSorter. It is taken from a popular open-source project Eurobudget [5] in the SourceForge website. Eurobudget is an office application written in Java, similar to Microsoft Money or Quicken.

Table 1 specifies the statistics of the three target programs. The sizes of these programs are in line with the sizes of the target programs used in typical software testing researches such as [1] or the famous Siemens test suites. 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.

Faulty Versions of Target Programs: To investigate the relative effectiveness of metamorphic testing and assertion checking, we used mutation operators to seed faults to programs. A previous study [1] showed that well-defined mutation operators were valid for testing experiments [6].

In our experiment, mutants were seeded using the tool muJava [26]. The tool uses two types of mutation operator: class

[3] Available at http://sourceforge.net/projects/jboolexpr.

[4] URL http://www.sourceforge.net.

[5] Available at http://eurobudget.sourceforge.net.

[6] We also attempted to use publicly accessible real faults of these programs to conduct the experiments. However, descriptions of these faults in the source repositories were either too vague or not available.

Table 2: Categories of Mutation Operators

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

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 [29] are operators specific for statement faults. We only seeded method level mutation operators to the programs under study, because our experiment concentrated on unit testing and because this set of operators had been studied extensively [29, 1]. Table 2 list all the mutation operators used in the controlled experiment.

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 used all of them in the controlled experiment.

4.3 Experimental Procedure

Before the experiment, the subjects were given a six-hour training to use MT and assertion checking. 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 mutants in the controlled experiment. For each program, the subjects were required to apply MT strictly following the procedure 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. The subjects were told to develop metamorphic relations and assertions as they saw fit, 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, for 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 the student submissions carefully to ensure that there was no implementation error.

4.4 Addressing the Threats to Validity

We briefly 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 conservative. 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.

External Validity: External validity is the degree to which the results are generalizable to the testing of real-world systems. The programs used in our experiment were 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.

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 Metamorphic Relations and Assertions

A critical and difficult step in applying MT and assertion checking is to develop metamorphic relations and assertions for 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 different 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. For the sake of brevity, we list in Table 5 only the metamorphic relations identified by the subjects for the Boyer program.

The results show that all the subjects could properly apply metamorphic testing and assertion checking after training. In general, they could identify a larger number of assertions than metamorphic relations. Furthermore, their abilities to identify metamorphic relations varied.

In particular, we observe that all 38 subjects managed to propose metamorphic relations after some training for each of the three open-source programs. It confirms the belief by the originators of MT that testers can formulate metamorphic relations effectively.

5.2 Comparison of 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 [1] 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. For metamorphic testing, a mutant is detected if a source test case and follow up test cases executed

Table 3: Number of Mutants by Operator Category

	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

Table 4: Number of Mutation Operators and Assertions

Class	Metamorphic Relations					Assertions			
	Total	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

Table 5: Metamorphic Relations for Boyer Program

	Description
1	$(x_2 = x_1 + s)$ & $(y_2 = y_1)$ & $(f(x_1, y_1) > -1) \Rightarrow f(x_1, y_1) = f(x_2, y_2)$
2	$(x_3 = x_1 + x_2)$ & $(y_1 = y_2 = y_3)$ & $(f(x_1, y_1) = -1) \Rightarrow f(x_3, y_3) \leq f(x_2, y_2) + len(x_1)$
3	$(x_1 = x_2)$ & $(y_2 = y_1.substr(0, len(y_1) - 1)) \Rightarrow f(x_2, y_2) \leq f(x_1, y_1)$
4	$(x_3 = x_1 + x_2)$ & $(y_1 = y_2 = y_3)$ & $(f(x_1, y_1) = -1)$ & $(f(x_2, y_2) > -1) \Rightarrow f(x_3, y_3) \leq f(x_2, y_2) + len(x_1)$
5	$(x_2 = upperlowercase(x_1))$ & $(y_2 upperlowercase(y_1)) \Rightarrow f(x_1, y_1) = f(x_2, y_2)$
6	$(x_2 = offset(x_1))$ & $(y_2 offset(y_2)) \Rightarrow f(x_1, y_1) = f(x_2, y_2)$
7	$(x_1 = x_2)$ & $(y_2 = y_1.substr(0, len(y_1)/2)) \Rightarrow f(x_2, y_2) \leq f(x_1, y_1)$
8	$(x_2 = x_1.substr(1, len(x_1) - 1))$ & $(y_1 = y_2))$ & $(f(x_1, y_1) \geq 1) \Rightarrow f(x_2, y_2) = f(x_1, y_1) - 1$
9	$(x_2 = x_1 + y_1)$ & $(y_1 = y_2))$ & $(f(x_1, y_1) = -1) \Rightarrow f(x_2, y_2) \geq len(x_1) - len(y_1)$
10	$(x_2 = reverse(x_1))$ & $(y_2 = reverse(y_1)))$ & $(f(x_1, y_1) > -1) \Rightarrow f(x_2, y_2) > -1$
11	$(x_2 = x_1.substr(0, f(x_1, y_1))$ & $(y_1 = y_2)$ & $(f(x_1, y_1) > -1) \Rightarrow f(x_2, y_2) = -1$
12	$(x_2 = x_1.substr(f(x_1, y_1), len(x_1) - f(x_1, y_1)))$ & $(y_1 = y_2))$ & $(f(x_1, y_1) > -1) \Rightarrow f(x_2, y_2) = 0$
13	$(x_2 = reverse(x_1))$ & $(y_2 = reverse(y_1))$ & $(f(x_1/y_1, y_1) = -1) \Rightarrow f(x_1, y_1) + f(x_2, y_2) + 2 * len(y_1) \geq len(x_1)$
14	$(x_2 = duplicate(x_1))$ & $(y_2 = duplicate(y_1))$ & $(f(x_1, y_1) > -1) \Rightarrow f(x_2, y_2) = f(x_1, y_1) * 2$
15	$(x_2 = x_1.substr(0, f(x_1, y_1) + len(y_1))$ & $(y_1 = y_2))$ & $(f(x_1, y_1) > -1) \Rightarrow f(x_2, y_2) = f(x_1, y_1)$
16	$(x_2 = reverse(x_1))$ & $(y_2 = reverse(y_1))$ & $(f(x_1, y_1) > -1) \Rightarrow f(x_1, y_1) + f(x_2, y_2) + len(y_1) \leq len(x_1)$
17	$(x_2 = reverse(x_1))$ & $(y_2 = reverse(y_1)))$ & $(f(x_1, y_1) = -1) \Rightarrow f(x_2, y_2) = -1$
18	$(x_2 = x1.substr(0, f(x_1, y_1) + len(y_1)))$ & $(y_1 = y_2)$ & $(f(x_1, y_1) > -1) \Rightarrow f(x_2, y_2) = len(x_2) - len(y_2)$
The function $f(x, y)$ returns the index of pattern y within the string x if x contains y; otherwise, return -1. (x_1, y_1), (x_1, y_1), and (x_3, y_3) are test cases for the function $f(x, y)$. $x_1, x_2, x_3, y_1, y_2, y_3$, and s are arbitrary strings. $x + y$ appends string y to string x. x/y removes string y from string x. $x.substr(m, n)$ returns a new string that is a substring of x. $len(x)$ returns the length of string x. $upperlowercase(x)$ changes each character in string x from lower case to upper case and vice versa. $offset(x)$ offsets each character in string x to get a new string. $reverse(x)$ reverses the string x. $duplicate(x)$ duplicates each character in string x to get a new string.	

Table 6: Mutation Detection Ratios for Metamorphic Testing and Assertion Checking

Class	Metamorphic Testing					Assertion Checking					p-Value
	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

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 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. They were *significantly* better than the corresponding mutation detection ratios for assertion checking. The p-value of the 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.

Our empirical results show that metamorphic testing is effective for object-oriented testing. On the other hand, there is a need to propose more systematic methods for creating metamorphic

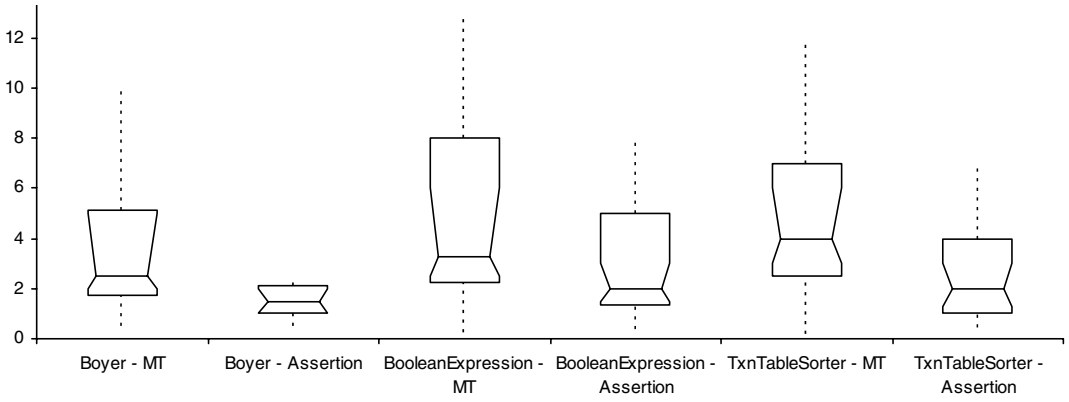

Figure 2: Box-and-Whisker Plots of Time Costs for Applying Metamorphic Testing and Assertion Checking

relations and assertions, since 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.

5.3 Comparison of Efforts

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.

Figure 2 shows box-and-whisker plots of the time costs 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 metamorphic relations. The time cost for assertion checking includes the time to add assertions to the source code.

The vertical axis of the figure 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.

The difference in time cost is acceptable, because the majority of the subjects have had prior experience in assertion checking. Figure 2 also indicates that the time cost for applying MT to object-oriented testing 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 effort for preparation.

5.4 Further Discussions on MT

In general, we observe that the more MRs being used, the higher would be the mutation detection ratio. As we have indicated in Section 5.2, there is a need to propose more systematic methods for creating metamorphic relations. The effectiveness of using an MR also increases as we increase the number of test cases. Since software testing resources are limited in real-life, it is also worth investigating on the number of test cases adequate for MT.

Furthermore, it is desirable to know which MRs should be given a higher priority. We analyzed the experimental results of Boyer program as an example. A total of 18 metamorphic relations were identified for this program. The fault detection capability of each MR was different. In particular, four subjects only identified metamorphic relation 1 in Table 5. The mutation detection ratios resulting from the work by these subjects were no more than 60% no matter how many test cases they used. Other subjects using metamorphic relation 4 or 12 in Table 5 achieved mutation detection ratios higher than 80%, although some of them only proposed four source test cases. It indicates that the quality of MR can be a key factor in determining the effectiveness of MT. More investigations are vital.

6. CONCLUSION

This paper reports on a controlled experiment to study the application of metamorphic testing (MT). The main objective is to evaluate whether MT is a useful and viable strategy and to assess the cost and effectiveness of MT. 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 an effective testing strategy in terms of fault detection capability. The time cost for applying MT is acceptable compared with assertion checking.

7. REFERENCES

[1] 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, 2005.

[2] 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, 1993.

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

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

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

[6] 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.

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

[8] 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.

[9] 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.

[10] 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, California, 2005.

[11] 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*, to appear.

[12] 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.

[13] 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, California, 2005.

[14] 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*, to appear.

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

[16] 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, California, 2002.

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

[18] 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.

[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*). ACM Press, New York, 2006.

[20] 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, California, 2004.

[21] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *IEEE Transactions on Software Engineering*, 19 (8): 774–787, 1993.

[22] 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, California, 2003.

[23] 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.

[24] 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, 2003.

[25] 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*). ACM Press, New York, 2006.

[26] 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.

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

[28] A. M. Memon, M. E. Pollack, and M. L. Soffa. Automated test oracles for GUIs. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2000/FSE-8*), pages 30–39. ACM Press, New York, 2000.

[29] 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.

[30] 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, California, 2003.

[31] 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, 2004.

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

[33] 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, California, 2004.

[34] 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.

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

[36] Q. Xie and A. Memon. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology*, to appear.