**Abstract**

Metamorphic testing is a software testing technique which can effectively alleviate the oracle problem. The main idea of metamorphic testing is to test a software by checking whether a metamorphic relation holds among several executions. How to acquire metamorphic relations from the software under test is a very essential task in metamorphic testing. METRIC (Metamorphic Relation Identification Based on Category-choice Framework) is a systematic technique to help software testers acquire metamorphic relations. This technique takes the input of software under test into consideration and uses candidate pairs formed by complete test frames to identify metamorphic relations. This paper believes that the output is an important aspect of a software and should also be considered during a test procedure. In this paper, we proposed an improved approach named as METRIC\* to enhance METRIC through introducing the output of software. A tool implementing METRIC\* is developed. Four real-life specifications are engaged in empirical study and the results show that METRIC\* can effectively reduce unusable candidate pairs and thus improve the efficiency of METRIC. The fault detection effectiveness of metamorphic relations identified through METRIC\* is investigated and the results shows that these metamorphic relations have good fault detection capability.

1. **Introduction**

Software testing is an important approach to guarantee the quality of software systems. When quality of software systems refers to the correctness of functions, we test whether the software can do the right thing in a right way. Under this circumstance, we need to verify the correctness of the software outputs. For example, we can compare the actual output with expected output to achieve the above purpose. The mechanism for determining a test has passed or failed is called an oracle [1]. For many software testing techniques, it is assumed that the oracle of software under test exists. But when it comes to the case that oracle does not exist, these testing techniques is hard to be applied.

Metamorphic testing [2] is a software testing technique that aims at situations when oracle does not exists. This technique uses metamorphic relations which can be derived from the characteristics of a software to detect faults. For a given software, we can derive some essential properties according to its functions that reflect the inner relationship among several tests. These properties are commonly known as metamorphic relations and is expected to be hold between the function and its corresponding implementation. Otherwise, the implementation is said to be faulty. Throughout the testing process, expected output is not involved and oracle problem is alleviated.

Given the advantage of metamorphic testing, more and more researchers and software testers are beginning to focus on this promising testing technique. Metamorphic testing has now been successfully used in various fields (domains) and has detected some real faults which have never been found before. It is reported that with a small number of diverse metamorphic relations, metamorphic testing could effectively help alleviate the oracle problem [3].

While metamorphic testing is receiving more and more attentions, there are still some shortcomings, making its development and application hampered. Among these shortcomings, metamorphic relation identification is a challenging task in metamorphic testing. From the perspective of software under test, metamorphic relations can reflect those inner and underlying characteristics of software which is hard to be recognized through test cases. From the perspective of software testers, metamorphic relations are used to detect faults and act as a kind of test oracle.

Metamorphic relations play an important role in metamorphic testing, without which it is not able to verify the correctness of a software through metamorphic testing. But in reality, metamorphic relation identification is done manually and even experts may find it difficult to acquire metamorphic relations [4]. If metamorphic relations cannot be identified or generated effectively and efficiently, metamorphic testing is hard to be applied.

To address this problem, Chen et al. developed a systematic and effective metamorphic relation identification methodology named as METRIC [5]. METRIC uses complete test frames generated by category choice framework [6] according to the input of a software to help software testers obtain metamorphic relations in a systematic way. A tool named MR-GEN is developed and software testers can make use of it to identify metamorphic relations. This methodology enable testers to form candidate pairs using complete test frames and decide whether these candidate pairs can be used to derive metamorphic relations. (If a candidate pair is useful for MR identification, then software testers can make use of it to acquire the inner metamorphic relations. If a candidate pair is useless for MR identification, then it is abandoned.)

METRIC takes the input of software into consideration. But as for a software, not only the input is an important perspective, the output is also an important perspective. Sometimes it is not comprehensive to consider the input domain only. Huai et al. [7] discovered that partition testing could be enhanced by considering the variation of output. Intuitively speaking, METRIC which is built upon category choice framework, a partition testing technique, could be enhanced by introducing software output. In this paper, an improved approach named as METRIC\* has been proposed to enhance METRIC mainly through introducing software output. A supporting tool named MR-GEN\* is developed, which automates parts of METRIC\* and helps testers with metamorphic relations identification. Empirical study engaging four real-life specifications is conducted to study the feasibility of METRIC\* in terms of metamorphic relation identification. The results shows that METRIC\* can be practically used for MR identification. Furthermore, fault detection effectiveness of metamorphic relations identified through METRIC\* is investigated. The results indicates that metamorphic relations identified through METRIC\* have better fault detection effectiveness compared with metamorphic relations identified through μMT [8].

The rest of this paper is organized as follows: Section 2 introduces the concepts of metamorphic testing, category-choice framework and METRIC. Section 3 explains the motivations of introducing output and methodology of METRIC\*. Section 4 describes the details of a supporting tool named as MR-GEN\*. Section 5 describes the setting of empirical study. Section 6 reports the results and discussions of empirical study. Section 7 discusses limitations and potential threads to the study. Section 8 summarizes this paper.

**2 Background Knowledge**

**2.1 Metamorphic Testing**

Metamorphic testing was proposed by Chen et al. in 1998, which is still applicable in absence of oracle when testing a software. The basic steps of metamorphic testing are as follows:

1. Identify necessary properties of the software under test and derive relations between multiple inputs and their corresponding outputs, commonly known as metamorphic relations.
2. Generate test cases using existing test case generation techniques. These test cases are referred to as source test cases.
3. Generate follow-up test cases from source test cases according to metamorphic relations derived before.
4. Apply source test cases and their corresponding follow-up test cases to the software under test and check whether a metamorphic relation holds among the outputs of relation’s relevant source test cases and follow-up test cases.

The following example illustrates the detailed process of metamorphic testing.

A

E

B

C

D

10

7

5

8

1

3

2

4

6

Figure 1 An Input Graph for Example 1

Example 1(Shortest Path in an Undirected Graph). Consider a program SP that can search for the shortest path between two specific nodes in a given undirected graph like Figure 1. We use SP(G,A,B) to represent the shortest path found from point A to point B in graph G. When graph G is complicated, it is not easy to verify whether the correctness of program output. But there are some properties within program SP that we can make use of. It is very easy to know that if we swap the start node and the end node, the length of shortest path between two nodes will not change. This property can be used to derive the changes of output when input is changed in a specific way, thus it can be considered as a metamorphic relation. If we construct test cases that satisfy the above differences in terms of input, then the output of these test cases should theoretically be the same. If we apply these test cases to the program under test and find out that the outputs of these test cases are not the same, then we have confidence that the program under test is faulty. Consider a source test case (G, A, B). According to the metamorphic relation, follow-up test case (G, B, A) is constructed. After applying these two test cases to execute the program, we decide whether |SP(G, A, B)|equals to |SP(G, B, A)|. If metamorphic relation is not satisfied, then program SP is faulty. Another property within program SP which we can make use of is that if point C is a node in the shortest path between point A and point B, then the length of shortest path from A to B must be the sum of shortest distance from A to C and from C to B. The inner metamorphic relation can be described as follow: |SP(G, A, B)| = |SP(G, A, C)| + |SP(G, C ,B)| where C is a node in SP(G, A, B). According to this relationship we can derive two follow-up test cases, (G, A, C) and (G, C, B), from (G, A, B). After the execution of these test cases, the output can be verified by checking whether above metamorphic relation is satisfied.

Above example shows that expected output of test cases were not involved in the whole testing procedure, thus metamorphic testing can alleviate the oracle problem and can effectively test programs when oracle problem occurs.

Commonly, a metamorphic relation is consist two parts: a sub-relation on input and a sub-relation on output. As for the first property within the example above, the sub-relation on input, denoted as “r”, can be described as follow: the start point and the end point is swapped. The sub-relation on output, denoted as “rf”, can be described as follow: the length of shortest path do not change.

**2.2METRIC**

Normally, when identifying a metamorphic relation, we need to consider how the output changes when the input is changed. Simply speaking, we need to change software inputs first and then get the corresponding predictable change in output. So how to fulfill the above objective? One way is to change inputs directly and then predicate changes in output according to specifications, which is simple but undirected. Another way is to make comparison between different inputs, and predict changes in terms of output. But this will also bring problems. How to acquire different inputs is an essential task. A common approach is to generate different test cases using existing test case generation method, but this will bring another problem. A large number of test cases might be generated and a number of comparisons need to be done, which is not feasible. The category-choice framework can generate complete test frames from which test cases can be generated. Each complete test frame represents an input scenario and therefore more abstract than test cases. It is feasible to compare test frames rather than compare test cases directly. Based on the above ideas, Chen et al. proposed metamorphic relation identification based on category choice framework (METRIC). METRIC is a systematic methodology which enables software testers to identify metamorphic relations. This technique makes use of complete test frames generated by category-choice framework according to software specifications and guide software testers in an easy and brief way.

The basic steps of METRIC are as follows:

1. Two distinct complete test frames are selected to form a candidate pair.
2. Enable the software tester to determine whether the current candidate pair implies a metamorphic relation. If so, enable the software tester to describe the inner metamorphic relation.
3. Restart from step 1, and repeat until all the complete test frames are compared with any other complete test frames or the expected number of identified metamorphic relations is reached.

Candidate pairs that implies metamorphic relations are said to be usable, and those that do not implies metamorphic relations are said to be unusable.

METRIC enable testers to concentrate on only one pair of complete test frame and determine the relationship between the outputs, which is far easier than the traditional way of considering both input and output aspects.

**2.3 CHOCLATE-DIP**

Choice relation framework with distinguishing output scenarios (CHOC’LATE-DIP) [7] is a partition testing technique based on CHOC’LATE. By introducing the output scenario, CHOC’LATE-DIP improves CHOC’LATE from the following perspectives:

1. Different scenarios of program outputs are identified and refined into categories and choices. These categories and choices related to program outputs are referred to as O-categories and O-choices respectively. Categories and choices which related to program inputs are referred to as I- categories and I-choices.
2. Choice relation table including I-choices and O-choices is constructed. This extended choice relation table not only contains the relation between each I-choice pair, but also contains relation between each O-choice pair as well as that between every I-choice and O-choice.
3. Complete test frames containing I-choices and O-choices are constructed based on extended choice relation table. These complete test frames which contains information about program output are referred to as IO-based complete test frame (IO-CTF).
4. When generating test cases from IO-CTF, the type of expected output can also be determined. This advantage makes it easier for software testers to determine the expected output corresponding to a test case.

As a reminder, a IO-CTF composed of input choices including “a”, “b”, and a output choice “c” is denoted as “{a, b; c}” in this paper.

**3 METRIC\* methodology**

**3.1 Motivation**

Partition testing has been receiving increasing attention for its simplicity and usability. A common approach is to divide all possible program inputs into disjoint partitions, from which test cases are selected. The idea is that all test cases within a partition shall be homogenious. So theoretically, a single test case will be sufficient to represent a partition. But when the partition is not homogeneous, then the statement above will be faulty. Huai et al. discovered that partition testing could be enhanced by considering the variation of output. Their studies have shown that if output is introduced in complete test frame generated by category-choice framework, the partition can be further subdivided according to different outputs.

Based on the above observation, we can get an intuition that METRIC, built upon category-choice framework, can be enhanced through introducing software output into complete test frames. Simply speaking, when identifying metamorphic relations, the use of IO-CTF has a great advantage than original complete test frames. The reasons are as follows:

1. IO-CTF contains information related to the output compared to the original complete test frame. When determining candidate pairs, software testers can directly see the changes that occur in output rather than predicting those changes through original complete test frames.
2. The introduction of output enables testers to group complete test frames. IO-CTFs are classified into several groups of which the combination of output choices are the same. After IO-CTFs are grouped, it is possible to reduce unusable candidate pairs. As shown in Figure 2, suppose group A has 3 IO-CTFs and their corresponding output is “output1”, group B has 3 IO-CTFs and their corresponding output is “output2”. Select a complete test frame from group A and select a complete test frame from group B to form a candidate pair. When all combinations are exhausted, nine candidate pairs are generated. If METRIC is used to identify metamorphic relations, all nine candidate pairs need to be decided. When output is introduced and IO-CTFs are grouped, we first compare group A with group B and decide whether we can derive output relations through comparing “output1” and “output2”. If not, then it is obvious that these nine candidate pairs are not useable for metamorphic relation identification, thus unusable candidate pairs are abandoned and total number of candidate pairs that need to be decided is reduced. Actually, software output is used to extract the commonality within candidate pairs and makes it possible to identify unusable candidate pairs.

Group1

Input 1

Input 2

Output 1

Output 1

Input 3

Output 1

TF1:

TF2:

TF3:

Group2

Input 4

Input 5

Output 2

Output 2

Input 6

Output 2

TF4:

TF5:

TF6:

Output 1

Output 2

?

Figure 2 Test Frame Groups With Different Outputs

1. More automatic preparation and systematic guidelines are introduced at the same time. The grouping of IO-CTFs and the reduction of candidate pairs can be done in a more automatic way.

**3.2 Methodology**

Through introducing extra O-categories and O-choices, the identification of metamorphic relations is more directly. Furthermore, it is possible to group IO-CTFs and reduce unusable candidate pairs.

In general, METRIC\* involves the following tasks:

1. Construct the complete test frames (IO-CTF), containing both I-category/I-choice and O-category/O-choice, that is, each IO-CTF is a possible combination of I-choices and O-choices;
2. Classify all IO-CTFs into groups of which the O-choice combinations are the same. These groups are called test frame groups. Within a test frame group, all IO-CTFs have the same combination of O-choices.
3. Identify metamorphic relations through candidate pairs formed by IO-CTFs with the same O-choice combination:

Select a test frame group and enable software tester to determine whether the output relation, denoted as Ro, exists when the O-choice combination remains unchanged.

* If Ro exists and the number of IO-CTFs belonging to current test frame group is more than 1, then select two IO-CTFs within current test frame group that are distinct in the combination of I-choices to form a candidate pair for user’s consideration. Enable the tester to determine the input relation, denoted as Ri, between the I-choice combinations of two IO-CTFs. Finally a metamorphic relation is defined as Ri plus Ro. If current test frame group has only one IO-CTF, then change another group and restart from task 3. Repeat this step until every IO-CTF has combined with any other IO-CTF within current test frame group.
* If Ro does not exist, then change another group and restart from task 3.

1. Identify metamorphic relations through candidate pairs formed by IO-CTFs with different O-choice combinations:

Select two test frame groups and enable the tester to determine whether the output relation, denoted as Ro, exists between O-choice combinations of these two test frame groups.

* If Ro exists, then select two IO-CTF, each of which is from one of the test frame groups, respectively, as a candidate pair for tester’s consideration. Enable the tester to determine the input relation, denoted as Ri, between I-choice combinations of these two IO-CTFs. Finally a metamorphic relation is defined as Ri plus Ro. Repeat this step until every IO-CTF within a test frame group has combined with every IO-CTF within another test frame group.
* If Ro does not exist, then change another two groups and restart from task 4.

When Ro does not exist in task 3, it indicates that the change of output cannot be determined. Thus the output relation of candidate pairs formed by any two IO-CTFs within current test frame group cannot be determined, which means that these candidate pairs are unusable. Then there is no need to select two IO-CTFs within this test frame group and form candidate pairs. In task 4, the reason is the same. Simply speaking, unusable candidate pairs are identified and abandoned in advance. As for METRIC, a tester has to determine every candidate pair and determine whether they are useful.

An example is proposed to demonstrate how METRIC\* works.

Example2 (Credit Card System). Consider a credit card system, which processes customer’s operation request and output whether the request is accepted or rejected. The input of this credit card system includes card number, status, manager’s special permission, credit limit and the amount of operation. The basic business process of the system are as follows:

1. Check whether the card number exists. If it exists, go to step 2. Otherwise, return “Rejected”.
2. Check the status of credit card. If the credit card is “activated”, go to step 4. Otherwise, go to step 3.
3. If the credit card has manager’s special permission, then return “Accepted”. Otherwise, return “Rejected”.
4. If the amount of operation is larger than credit limit, then return “Rejected”. Otherwise, return “Accepted”.

According to CHOC’LATE-DIP, we can acquire categories and choices shown in Table 1 and Table 2.

Table 1 Input Categories and Choices of Example 2

|  |  |
| --- | --- |
| I-category | I-choice |
| 1．Card number | 1a. exist  1b. do not exist |
| 2．Status | 2a. activated  2b. not activated |
| 3．Manager’s special permission | 3a. yes  3b. no |
| 4．Amount of operation | 4a. ≤ credit limit  4b. > credit limit |

Table 2 Output Categories and Choices of Example 2

|  |  |
| --- | --- |
| O-category | O-choice |
| I．Return | Ia. Accepted  Ib. Rejected |

After identifying categories and defining choice, IO-CTFs are generated through combining choices from different categories. All IO-CTFs generated using CHOC’LATE-DIP are shown below.

Table 3 IO-based Complete Test Frames

|  |  |  |  |
| --- | --- | --- | --- |
| {1a,2a,3a,4a;Ia} | {1a,2a,3a,4b;Ib} | {1a,2a,3b,4a;Ia} | {1a,2a,3b,4b;Ib} |
| {1a,2b,3a,4a;Ia} | {1a,2b,3a,4b;Ia} | {1a,2b,3b,4a;Ib} | {1a,2b,3b,4b;Ib} |
| {1b,2a,3a,4a;Ib} | {1b,2a,3a,4b;Ib} | {1b,2a,3b,4a;Ib} | {1b,2a,3b,4b;Ib} |
| {1b,2b,3a,4a;Ib} | {1b,2b,3a,4b;Ib} | {1b,2b,3b,4a;Ib} | {1b,2b,3b,4b;Ib} |

Next, IO-CTFs are classified into groups of which the O-choice combinations are the same. Test frame groups and their corresponding IO-CTFs are shown below.

Table 4 IO-CTF Groups

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Group Name | IO-CTF | | | |
| Group Ia | {1a,2a,3a,4a;Ia} | {1a,2a,3b,4a;Ia} | {1a,2b,3a,4a;Ia} | {1a,2b,3a,4b;Ia} |
| Group Ib | {1a,2a,3a,4b;Ib} | {1a,2a,3b,4b;Ib} | {1a,2b,3b,4a;Ib} | {1a,2b,3b,4b;Ib} |
| {1b,2a,3a,4a;Ib} | {1b,2a,3a,4b;Ib} | {1b,2a,3b,4a;Ib} | {1b,2a,3b,4b;Ib} |
| {1b,2b,3a,4a;Ib} | {1b,2b,3a,4b;Ib} | {1b,2b,3b,4a;Ib} | {1b,2b,3b,4b;Ib} |

We first identify metamorphic relations within each test frame group. It is obvious that two test frame groups need to be identified. We choose Group Ia first. The corresponding O-choice of Group Ia is “Accepted”. When the O-choice is “Accepted” and holds, it is easy to figure out that output do not change and remains “Accepted”. After that, we select two IO-CTFs from Group Ia and determine input relation. Suppose {1a,2a,3a,4a;Ia} and {1a,2a,3b,4a;Ia} are selected. Comparing the I-choice part, the input relation is determined as follow: The number of credit card exists, the status of credit card is activated and the amount of operation is under credit limit, but special permission is changed. Finally a metamorphic relation is defined as input relation plus output relation. Detail metamorphic relation is described as follow: In the case where the card number exists, the status of credit card is activated and the amount of operation is less than credit limit, when special permission is changed, the output will not change and remains “Accepted”. Repeat above steps until every IO-CTF has combined with any other IO-CTF within Group Ia. When dealing with Group Ib, processes are the same.

We identify metamorphic relations across two different test frame groups next. Two test frame groups are generated and thus it is only able to select these two test frame groups for tester’s consideration. The O-choice of Group Ia is “Accepted” while O-choice of Group Ib is “Rejected”. When the O-choice is changed, the output of program under test is changed. After that, select one IO-CTF from Group Ia select one IO-CTF from Group Ib to form a candidate pair for software tester’s consideration. Suppose {1a,2a,3a,4a;Ia} and {1a,2a,3a,4b;Ib} are selected. A tester can determine the input relation through comparing the I-choice part. The input relation is described as follow: the number of credit card exists, the status of credit card is activated and the credit card has manager’s special permission, but the amount of operation changes from no more than credit limit to higher than credit limit or the opposite. Finally a metamorphic relation is defined as input relation plus output relation. Detail metamorphic relation is described as follow: In the case where number of credit card exists, status of credit card is activated and credit card has manager’s special permission, when the amount of operation changes from no more than credit limit to higher than credit limit or the opposite, the output will change oppositely. Repeat above steps until every IO-CTF within group Ia has combined with every IO-CTF within Group Ib.

**4 Tool**

Section 4 describes the details of a supporting tool named as MR-GEN\*. Based on MR-GEN, a supporting tool called MR-GEN\* has been developed. New features including abilities to process output related categories and choices, group IO-CTFs, identify and abandon unusable candidate pairs have been integrated into the supporting tool.

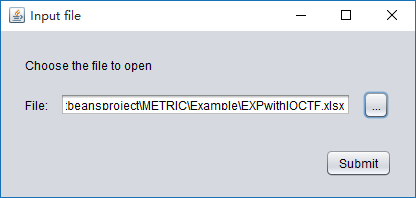


Figure 3 Screen for file selection

MR-GEN\* acquires categories, choices, and IO-CTFs of software under test through a spreadsheet. Data in this spread sheet is stored in a specific form which is easy for software testers to view and modify. MR-GEN\* allows the tester to specify the maximum number of metamorphic relations to be identified.

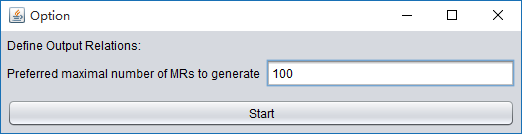


Figure 4 Screen for defining maximum number of MRs

After the spreadsheet is read and parsed, pairs of test frame groups with their corresponding O-choice combinations are shown one by one. Differences between two O-choice combinations are marked in red, helping software testers to determine whether an output relation is implied. If output relation exists, MR-GEN\* enables testers to write down the relation and record it. Current pair of test frame group will be marked as usable and relevant candidate pairs will be generated for input relation determination. If output relation does not exists. MR-GEN\* enables testers to mark this pair of test frame group as unusable and relevant candidate pairs will not be generated.

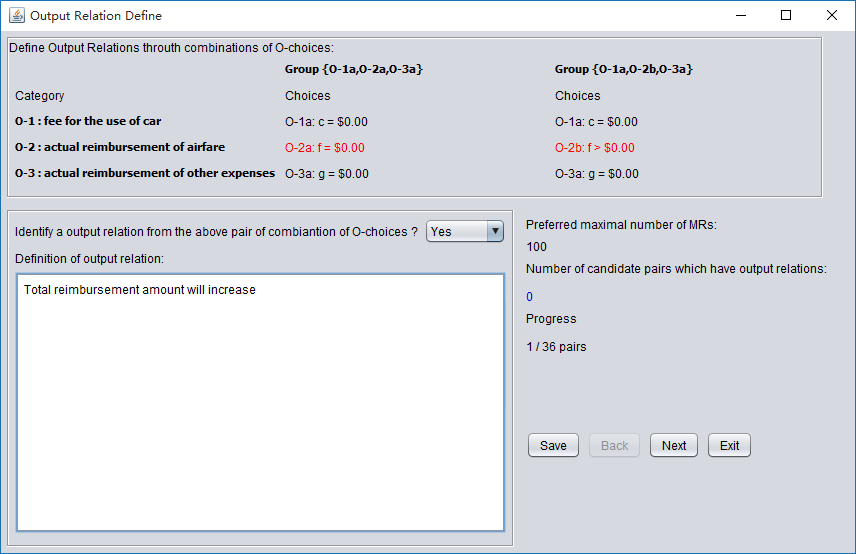


Figure 5 Screen for defining Output Relations

When all pairs of test frame groups are determined and candidate pairs are generated, all usable candidate pairs with their corresponding I-choice combinations are shown one by one. Also differences between two I-choice combinations are marked in red. Software testers can make use of the difference to determine the input relation that two I-choice combinations implies. MR-GEN\* enables testers to write down the input relation. When input relation of a candidate pair is defined, metamorphic relation is constructed by combining input relation and output relation. Testers can click “Display all MRs” to view the already defined metamorphic relations.

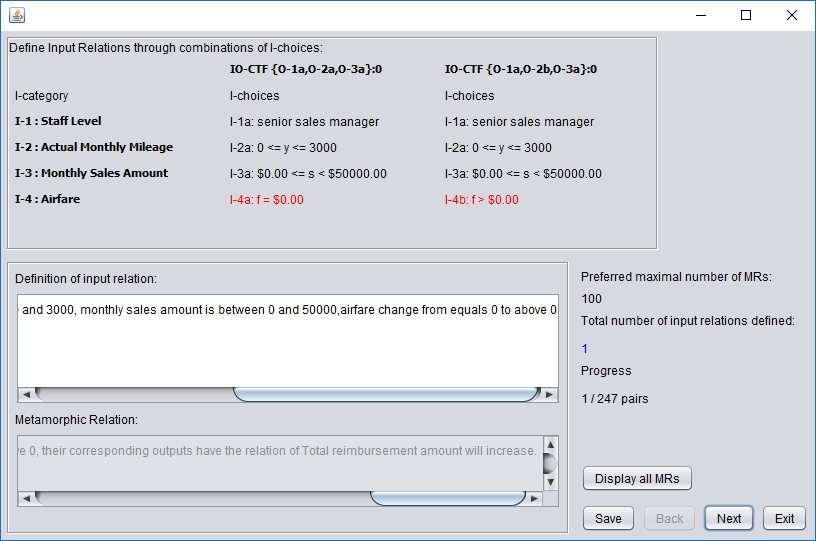


Figure 6 Screen for defining input relations

**5 Experimental settings**

**5.1 Research questions**

In section 3, the methodology of METRIC\* has been illustrated and the principle of improvement has been described. During the process of making changes to METRIC, we must have obtained something useful and abandoned something useless. Whether these changes have an impact on the original methodology in terms of metamorphic relation identification remains unknown. Whether METRIC\* can effectively reduce the number of candidate pairs that need to be decided is a problem that worthy of study. After identifying metamorphic relations, whether these relations are helpful for fault detection remains unknown. If these metamorphic relations can effectively detect faults, then METRIC\* is meaningful. Otherwise, METRIC\* will not be helpful for fault detection. Each method has its own shortcomings in terms of fault detection. It is a meaningful research to study the type of fault that is difficult to detect by metamorphic relations identified through METRIC\*, which can help us to discover the weakness of such metamorphic relations. Four research questions are settled as follows:

**RQ1: Can METRIC\* be practically used for MR identification?**

Even though METRIC\* and METRIC are very similar in terms of metamorphic relation identification mechanism, the differences between these two approach are also obvious. METRIC\* makes use of software output but METRIC don’t. Another change is that METRIC\* can group IO-CTFs according to O-choice combinations but METRIC don’t. These changes might weaken the ability of METRIC\* in terms of identification of metamorphic relations. So whether METRIC\* is able to identify metamorphic relations is a meaningful study.

Four commercial software specifications were used in the experiment. The goal is to observe whether METRIC\* can successfully identify metamorphic relations from these specifications. Participants involved in the experiment were graduate students whose major was software engineering.

**RQ2: Can METRIC\* reduce the number of candidate pairs that need to be decided?**

Theoretically, METRIC\* can identify and abandon unusable candidate pairs in advance, thus reduce the number of candidate pairs that need to be decided by software testers. But theory and practice are two different things. But in reality, whether METRIC\* can reduce the number of candidate pairs that need to be decided remains unknown.

To study this problem, it is essential to record the number of candidate pairs during the identification of metamorphic relations. Use “Num” to indicate the number of candidate pairs when using METRIC to identify metamorphic relations. Use “Num\*” to indicate the number of candidate pairs when using METRIC\* to identify metamorphic relations. The goal is to compare “Num \*” with “Num” and see whether the former is less than the latter.

**RQ3: How effectively can a metamorphic relation set which is formed by all metamorphic relations identified through METRIC\* detect software faults?**

Metamorphic relation is the key part of metamorphic testing. The fault detection capability of metamorphic relations set has a great influence on the effectiveness of metamorphic testing. To answer this research question, the fault detection effectiveness of metamorphic relations sets was studied through mutation analysis. The mutants killed by metamorphic relations sets are counted and mutation scores of metamorphic relation sets are calculated. A series test suits which are of different size are generated and used to study the impact of test suit’s size on fault detection capability of metamorphic relations set. Finally the fault detection capability of metamorphic relation sets identified through METRIC\* was compared with metamorphic relation sets identified through μMT.

**RQ4: Which faults are difficult to be detected by metamorphic relations identified through METRIC\*?**

By answering this research question, shortcomings of metamorphic relations identified through METRIC\* in terms of fault detection can be revealed. To study this question, mutants that could not be killed by all metamorphic relations were recorded and analyzed. The reasons why these mutants could not be killed were further investigated.

**5.2 Specification , Subject Program, and Mutant Generation**

As for RQ1, four commercial software specifications were used in the experiment. SPBC, which is the specification of a cellphone billing system related to a telecommunications company; SBBS, which is the specification of a baggage billing service related to an airline company; SCAR, which is the specification of a company car and expense claim system; and SMOS, which is the specification of a meal ordering system related to a company providing catering services for different airlines.

According to these specifications ,we have implemented Phone Bill Calculation(PBC), Baggage Billing Service(BBS), Car and Expense Claim System(CAR), and Meal Ordering System(MOS) respectively using Java programing language.

Phone bill calculation service is a mobile phone charge calculation system used in China Unicom. The purpose of phone bill calculation service is to determine a user’s phone charge within a month in terms of various kinds of aspects such as communication time, data usage, mobile phone tariffs. A user can calculate phone charge through the system.

Baggage billing service provide a passenger with baggage fee calculation service, with reference to China Airlines baggage accounting standards. A passenger can calculate his/her own baggage fees by offering relevant flight and baggage information including aircraft cabin, region, baggage weight, air fare and whether the passenger is a student to the service.

Car and expense claim system is an expense reimbursement system used in a large trading firm. CAR assists the sales director of the firm in determining the fee to be charged to each senior sales manager or sales manager for any “excessive” mileage in the use of the company car, and in processing reimbursement requests regarding various kinds of expenses such as airfare, hotel accommodation, meals, and phone calls. A sales staff uses CAR to make his claim at the end of each month.

Meal ordering system is used by an airline catering company to determine the quantity for every type of meal and other special requests (if any) that need to be prepared and loaded onto the aircraft served by the company. For each flight, MOS produces an output called “Meal Schedule Report” (MSR), which contains the following information: number of first-class meals, number of business-class meals, number of economy-class meals, number of meals for crew members, number of meals for pilots, number of child meals, and number of bundles of flowers.

Mutants for these four subject programs were generated using muJava [9]. Only method-level mutation operators were used to generating mutants, each of which contains only one fault. We generate 210, 187, 180, and 224 mutants for the four subject programs, respectively. Equivalent mutants are identified manually.

Table below provides a summary of specifications, subject programs and mutants.

Table 5 Sumary of specifications, subject programs and mutants

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Specification | Corresponding Subject Program | LOC | Mutation Operator Used | Number of All Mutants | Number of Equivalent Mutants |
| SPBC | PBC | 107 | AORB, AOIU, AOIS, ROR, COR, COI, LOI, SDL, VDL, ODL | 210 | 38 |
| SBBS | BBS | 97 | AORB, AOIU, AOIS, ROR, COI, LOI, SDL, VDL, CDL, ODL | 187 | 67 |
| SCAR | CAR | 117 | AORB, AOIU, AOIS, AOUD, ROR, COI, SDL, ODL | 180 | 29 |
| SMOS | MOS | 150 | AORB, AOIU, AOIS, LOI, SDL, CDL, ODL, JSI | 224 | 51 |

**5.3 Test Case Generation**

In this paper, the source test case and follow-up test case corresponding to a metamorphic relation were generated according to IO-CTFs from which this metamorphic relation is derived. Through deciding a candidate pair formed by two IO-CTFs, sub-relation on input and sub-relation on output are determined, from which metamorphic relation is constructed. So, the application domain of a metamorphic relation identified through METRIC\* is only the input and output field represented by these two IO-CTFs. When generating test cases for a metamorphic relation, it is necessary to select actual values in the input field to which the metamorphic relation applies.

For an IO-CTF, the input consists of multiple choices that belong to different categories. When generating a test case according an IO-CTF, it is necessary to take an value within the range corresponding to each choice that makes up the IO-CTF, then combine those values and form the test case. For a metamorphic relation, the generation of its corresponding test cases requires the generation of IO-CTF test cases from which this metamorphic relation is derived. One as source test case and the other as follow-up test case. These two test cases are put together as a test case for a metamorphic relation.

When selecting values from the range corresponding to a choice, we randomly generate values and select boundary values. Boundary value selection is only applied in source test case generation because if boundary value is selected in source test case generation and follow-up test case generation, there will be a combination explosion problem, resulting in excessive number of test cases.

The number of test cases generated for subject programs are shown below:

Table 6 Number of Test Cases

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Subject Program | Random | | | Boundary Value |
| 1 per MR | 5 per MR | 10 per MR |
| PBC | 142 | 710 | 1420 | 1096 |
| BBS | 735 | 3675 | 7350 | 3007 |
| CAR | 1130 | 5650 | 11300 | 7772 |
| MOS | 3512 | 17560 | 35120 | 50907 |

**6 Experimental Results and Discussions**

**6.1 Feasibility of METRIC\***

To answer RQ1, metamorphic relations identified by participants were counted and verified. Participants use the same complete test frames generated by CHOC’LATE-DIP and they have determined all candidate pairs. Results are shown in the table below.

Table 7 Number of identified metamorphic relations

|  |  |
| --- | --- |
| Specification | Number of correctly identified metamorphic relations |
| SPBC | 142 |
| SBBS | 735 |
| SCAR | 1130 |
| SMOS | 3152 |

The results have shown that METRIC\* can successfully identify metamorphic relations from all these specification, thus METRIC\* can be practically used for MR identification.

One further observation is that the metamorphic relation identified using METRIC is the same as using METRIC\*. The reason for this observation is as follow: complete test frames involved in METRIC are same as the I-choice combinations of IO-CTFs involved in METRIC\*. Output is the only difference between IO-CTF and complete test frame. It helps us to determine output relations in a more direct way. Thus, metamorphic relation identified using METRIC is the same as using METRIC\*.

**6.2 Reduction of candidate pairs**

RQ2 was answered by comparing “Num \*” with “Num”. If the former is less than the latter, then METRIC\* can successfully reduce the number of candidate pairs. If not, then the result needs further study. Results are shown in the table below.

Table 8 Number of candidate pairs and pairs of test frame groups

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Specification | Number of pairs of test frame groups | | | Number of candidate pairs | |
| Usable | Unusable | Total | Num\* | Num |
| SPBC | 2+36 | 10+30 | 12 + 66 | 142 | 496 |
| SBBS | 1+1 | 1+0 | 2+1 | 735 | 780 |
| SCAR | 1+4 | 7+24 | 8+28 | 1130 | 2145 |
| SMOS | 0+3512 | 0+12598 | 0+16110 | 3512 | 16110 |

In the above table and charts, the column with the heading “Number of candidate pairs” show the relevant statistics. For the four specifications, the numbers in column “Num\*” are less than the corresponding number in column “Num”. As for SPBC, number of candidate pairs is reduced to 142, less than one-third of the original. As for SBBS, the reduced amount of candidate pairs is small. As for SCAR, the number of candidate pairs is reduced to only half of the original. As for SMOS, the amount of reduced candidate pair is significant. Results indicate that METRIC\* can effectively identify and abandon unusable candidate pairs in advance and thus reduce the numbers of candidate pairs that need to be decided by software testers.

One observation is that the cost of using METRIC\* might be much more than using METRIC when identifying metamorphic relations from SMOS. The reason for this phenomenon is that there is only one IO-CTF in each test frame group. As shown in the figure below, if each test frame group has only one IO-CTF, then it’s not able to perform step 3 in METRIC\* which is to identify metamorphic relations within each test frame group. Under this circumstance, identifying metamorphic relations between two distinct test frame groups is equivalent to determining metamorphic relations within a candidate pair using METRIC. The reason is that IO-CTFs within these two test frame groups can only form one candidate pair. If we apply METRIC\* on this situation, output relation is first determined and then input relation is determined. Finally, a metamorphic relation is defined as input relation plus output relation. If we apply METRIC on this situation, the changes of input are considered and metamorphic relation is derived according to specifications. The former approach seems to be more complicated than the letter approach. Thus, when test frame groups contain a great number of groups which has only one IO-CTF, then it is preferred to use METRIC rather than METRIC\*. When most of the test frame groups contain more than one IO-CTF, then METRIC\* is preferred.

**6.3 Fault detection capability**

We conducted this experiment 10 times to reduce the impact of small probability events, making the results more convincing and reliable. The average mutation scores of test suites are shown in the table below. Note that column named as “Random” indicates that test suites corresponding to this column are generated randomly. “Boundary” indicates that test suites corresponding to this column are generated according to boundary conditions. “1 per MR”, “5 per MR”, and “10 per MR” indicates that test suites corresponding to these columns are of different sizes. Test cases that make up the test suites of column named “1 per MR” are generated 1 per MR. The other two columns are similar.

Table 9 Average Mutation Score

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Target Program | Average Mutation Score | | | |
| Random | | | Boundary |
| 1 per MR | 5 per MR | 10 per MR |
| PBC | 69.30% | 74.71% | 76.63% | 80.70% |
| BBS | 91.67% | 91.67% | 91.67% | 91.67% |
| CAR | 75.76% | 76.55% | 77.22% | 86.09% |
| MOS | 74.57% | 74.57% | 74.57% | 74.57% |

From the above table, we can observe that average mutation score ranges from 69.30% to 91.67%, which indicates that metamorphic relations sets identified through METRIC\* can detect most of the planted faults. This demonstrates good fault detection effectiveness in software testing. When comparing test suites of different sizes, we can observe that when the size of test suite becomes larger, the mutation score does not increase significantly. As for PBC, the mutation score only increased by 7.33%, while the size of test suite increased by 10 times of the original. As for BBS and MOS, the mutation scores of test suites with different sizes are all the same. This phenomenon indicates that increasing the size of a test suite cannot significantly improve the fault detection capability of its corresponding metamorphic relation set.

The distribution of mutation scores corresponding to experiments is shown below.

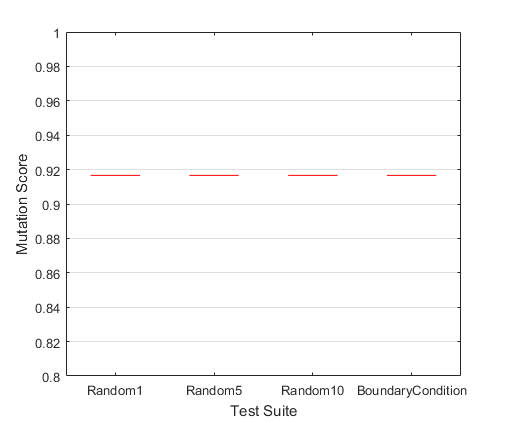
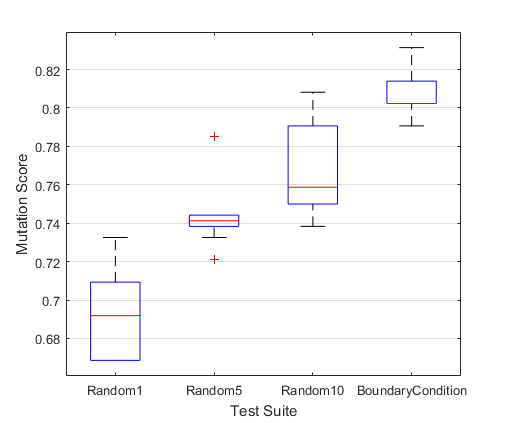


Figure 7 Mutation score distribution of PBC Figure 8 Mutation score distribution of BBS

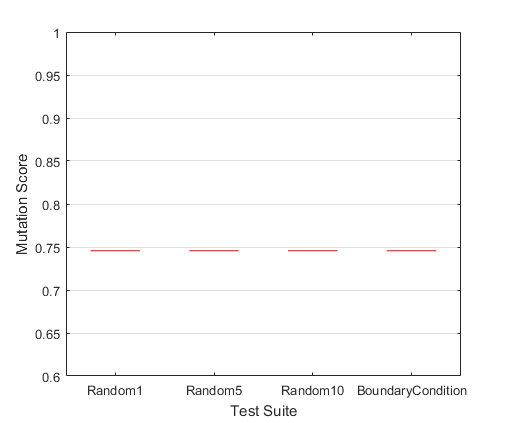
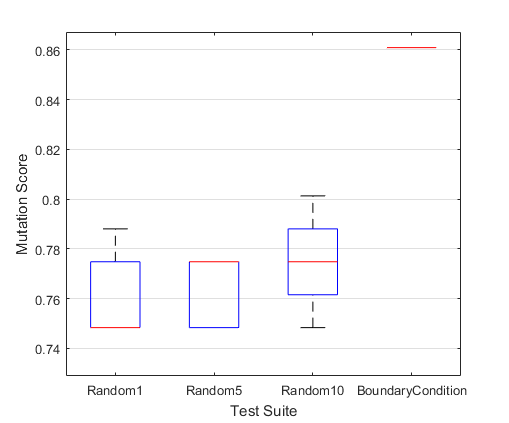


Figure 9 Mutation score distribution of CAR Figure 10 Mutation score distribution of MOS

From these four figures, we can observe that mutations scores corresponding to each experiment are distributed in small intervals, which means that the fault detection capability of metamorphic relation sets identified through METRIC\* is relatively stable.

We have also conducted experiments on studying the differences between metamorphic relations identified through METRIC\* and metamorphic relations derived by other techniques in terms of fault detections capability. We select μMT as contrast technique. We evaluated the differences in fault detections capabilities by comparing the mutation scores of metamorphic sets of the same size.

We applied μMT to these four subject programs and the numbers of metamorphic relations identified are shown below.

Table 10 Number of metamorphic relations identifided by μMT and METRIC\*

|  |  |  |
| --- | --- | --- |
| Subject Program | Number of MRs identified through | |
| μMT | METRIC\* |
| PBC | 32 | 142 |
| BBS | 36 | 735 |
| CAR | 60 | 1130 |
| MOS | 80 | 3152 |

An observation is that for these four subject programs metamorphic relations identified through μMT are less than metamorphic relations identified through METRIC\*. For the sake of fairness, we randomly select some of the metamorphic relations from the all metamorphic relations of METRIC\* to form a set that is the same size as the full metamorphic relation set of μMT. Suppose the subject program is PBC. When conducting an experiment, we randomly select a subset of 32 metamorphic relations from the full metamorphic relation set of METRIC\*. Then we use this subset to compare with the full metamorphic relation set of μMT in terms of mutation scores. Source test cases were generated 1 per metamorphic relations randomly within the input field of a metamorphic relation, thus the test suites corresponding to these two metamorphic relation sets are of the same size. Also this experiment is repeated ten times to reduce the impact of small probability events. The distribution of mutation scores is shown below.

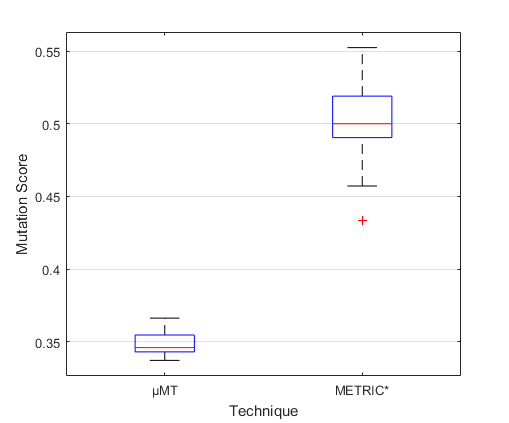
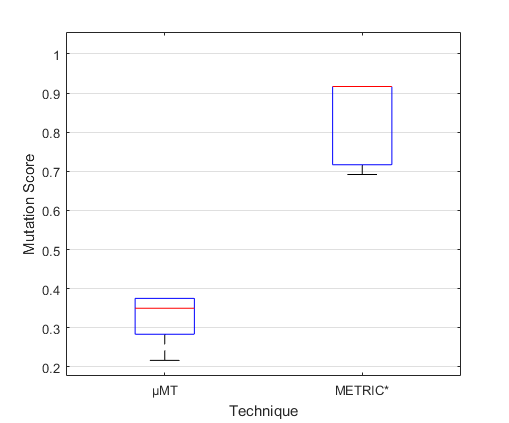
 

Figure 11 Mutation score distribution of PBC Figure 12 Mutation score distribution of BBS

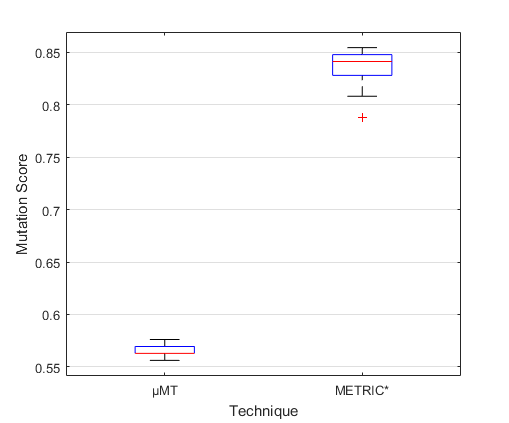
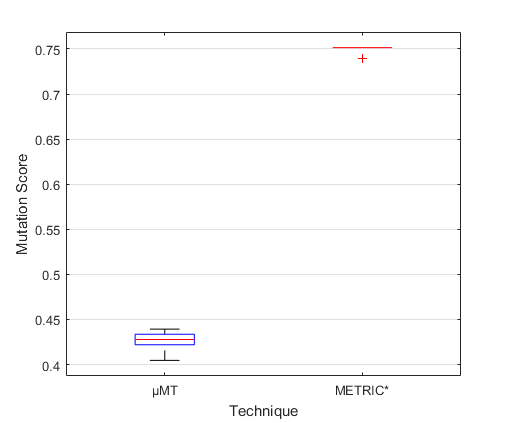
 

Figure 13 Mutation score distribution of CAR Figure 14 Mutation score distribution of MOS

From above figures, we can observe that the mutation scores of metamorphic relation set corresponding to METRIC\* is higher than that of metamorphic relation set corresponding to μMT, which means that METRIC\* produces a better metamorphic relation sets than the μMT produced.

**6.4 Non-killed mutants**

We have gathered statistics of mutants that are not killed and studied the characteristics of these mutants. Results are shown in the table below.

Table 11 Characteristics of non-killed mutatns

|  |  |  |  |
| --- | --- | --- | --- |
| Program | Mutation Operator | Number of non-killed mutants | Proportion |
| PBC | AOIS | 12 | 40.00% |
| AORB | 8 | 26.67% |
| ODL | 2 | 6.67% |
| SDL | 8 | 26.67% |
| VDL | 2 | 6.67% |
| BBS | AOIS | 1 | 10.00% |
| AORB | 4 | 40.00% |
| CDL | 3 | 30.00% |
| SDL | 3 | 30.00% |
| CAR | AOIS | 15 | 71.43% |
| ROR | 5 | 23.81% |
| SDL | 1 | 4.76% |
| MOS | AOIS | 14 | 32.56% |
| AORB | 15 | 34.88% |
| CDL | 6 | 13.95% |
| ODL | 6 | 13.95% |
| SDL | 2 | 4.65% |

From the above table, it is easy to find out that the mutants generated by mutation operator called AOIS account for the largest proportion of all non-kill mutants. This phenomenon drives us to study the reasons behind. We will use mutant called AOIS\_70 generated from program MOS to demonstrate why mutants produced by mutant operator AOIS are hard to be killed.

The fault planted in AOIS\_70 is at line 91. The correct statement :

this.msr.numberOfChildMeals = this.numberOfChildPassengers \* 2;

is changed into the following statement:

this.msr.numberOfChildMeals = --this.numberOfChildPassengers \* 2.

The difference between these two statements is that an operator “--” is added in front of the variable named “this.numberOfChildPassengers”, which let this variable be reduced by one before use. This statement is just before a statement which outputs the value “this.numberOfChildPassengers”, so this fault can be spread out.

We treat “this.msr.numberOfChildMeals” as the dependent variable and denote it with y . We treat “this.numberOfChildPassengers” as independent variable and denote it with x. Before the fault is planted, the relation between independent variable and dependent variable is as follow:

y = x\*2

After the fault is planted, the relation between x and y is as follow:

y = (x-1)\*2

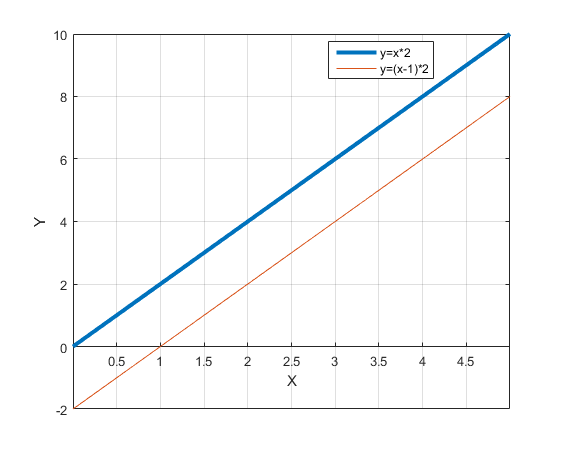


Figure 15 Graph of two functions

Through the above figure we can find that the planted fault simply moves line 1 down by two unit distances. The shape of line 1 is not changed, which means that the relative relation between any two points on this line has not changed. Metamorphic testing detects faults by checking whether the relations hold between several executions. As for the variable named “numberOfChildPassengers” in this example, metamorphic testing checks whether relation between two points on this line holds. As mentioned before, relation between any two points on this line has not changed, so this mutant is hard to be killed. Intuitively, if a fault planted cannot change the relations among output domain, then this fault is hard to be detected

**7 Limitations and Further work**

**7.1 Limitations**

Limitations of METRIC\* can be described from two aspects. The first respect is related to characteristics of test frame groups. When most of the test frame groups contains only one IO-CTF, then METRIC\* is not recommended to be applied. The reason is that identifying metamorphic relations between such test frame groups is equivalent to determining metamorphic relations within a candidate pair using METRIC. The second respect is related to the form of metamorphic relations. It is commonly known that a metamorphic relations is consist of a sub-relation on input and a sub-relation on output. But some metamorphic relations cannot be easily split into the above two parts. For example, the second metamorphic relation within Example 1 is not able to split into two parts.

This metamorphic relation is described as follow: |SP (G, A, B)| = |SP (G, A, C)| + |SP (G, C, B)| where C is a node in SP (G, A, B), which sub-relation on input and sub-relation on output are tangled together. If most metamorphic relations are of the first form that can be split into two parts, then METRIC\* can be used in most cases. Otherwise, the use of METRIC\* is limited. To address this problem, it is very essential to conduct a survey on the forms of metamorphic relations.

There are four limitations to the experiment of this study. The first limitation is that specifications involved in the experiment is simple. The number of categories identified from these specifications varies from 6 to 14 and total number of choices within these specifications varies from14 to 47. It is more appropriate to use larger specifications to verify the feasibility of METRIC\*. The second limitation is that it would be better to compare the efficiency between METRIC\* and METRIC. After all, these two methodology are used by software testers to identify metamorphic relations from software specifications. Although METRIC\* can reduce candidate pairs theoretically, in reality, there are still many factors that will affect the procedure of metamorphic relation identification. Theoretical analysis can only reflect the degree of improvement to a certain extent. The third limitation is that categories, choices, and complete test frames used for identifying metamorphic relations are pre-defined. In reality, categories and choices identified by different software testers are different and complete test frames generated would be totally different. This phenomenon could lead to different experiment results, which is worthy of further studies. The last limitation is that the set of source test cases corresponding to μMT and METRIC\* were not same, this may have impact on the results we have obtained. At first we had considered this problem and tried to construct source test cases that can be applied to metamorphic relations corresponding to these two techniques, but finally failed due to the variety of input field related to these metamorphic relations.

**7.2 Further work**

As for the survey on the forms of metamorphic relations, we will make use of METWiki [10] to conduct the survey. METWiki is a metamorphic relations repository, which enable testers to find desired metamorphic relations for reuse or reference. It is feasible to study metamorphic relations within this repository and figure out the forms of metamorphic relations. As for the limitations to experiment, an empirical study on comparing the efficiency between METRIC\* and METRIC will be conducted. Also, more complex specifications will be involved to enhance the credibility of the experiment.

**8 Related work**

Sun et al. proposed a data mutation directed metamorphic relation acquisition methodology called μMT. This methodology uses data mutation operators to identify DM-relations among input data of test cases and uses mapping rules of the program under test to identify the output relations of the related test cases. The feasibility of μMT and fault detection effectiveness of derived metamorphic relations were evaluated through three case studies. The result have shown that μMT is feasible to direct software testers to acquire metamorphic relations. Acquired metamorphic relations were found to be simple and demonstrated good fault detection effectiveness.

Kanewala and Bieman [11] proposed a metamorphic relation detection approach based on machine learning. This technique aims at detecting metamorphic relations from control flow graph of a system. A training set is required to create the predictive model. Once the training process is done, most parts of metamorphic relations detection process can be done automatically. Results of the experiment have shown that this approach is highly effective in predicting metamorphic relations under certain circumstances and predicted metamorphic relations can effectively detect faults.

Liu et al.[12] proposed a metamorphic relation composition method which aims at constructing metamorphic relations from existing ones. The intuition is that when a series of various metamorphic relations are composited, the new metamorphic relation is supposed to inherit all characteristics of old metamorphic relations. An experimental study was conducted and the results showed that metamorphic relations constructed through the composition of existing metamorphic relations can effectively improve the cost-effectiveness of metamorphic testing.

**9 Conclusion**

In this paper, we have proposed an enhanced approach named METRIC\* to identify metamorphic relations based on METRIC. We innovatively introduced software output to the original METRIC and formed a new methodology. METRIC\* considers both input and output of software under test to help software testers identify metamorphic relations, which is the key part of metamorphic testing. A supporting tool named MR-GEN\* is implemented, which automates parts of METRIC\*. Experiments are conducted and results have shown that METRIC\* is feasible for identifying metamorphic relations. Metamorphic relations identified through METRIC\* have shown good fault detection capability, which implies that METRIC\* is able to identify good metamorphic relations and thus can improve the effectiveness of metamorphic testing.

**References**

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

[2] Chen T Y, Cheung S C, Yiu S M. Metamorphic testing: A new approach for generating next test cases[R]. Technical Report HKUST-CS98-01, Dept. of Computer Science, Hong Kong Univ. of Science and Technology, 1998.

[3] Liu H, Kuo F C, Towey D, et al. How Effectively Does Metamorphic Testing Alleviate the Oracle Problem? [J]. IEEE Transactions on Software Engineering, 2014, 40(1):4-22.

[4] Groce A, Kulesza T, Zhang C, et al. You are the only possible oracle: Effective test selection for end users of interactive machine learning systems [J]. IEEE Transactions on Software Engineering, 2014, 40(3): 307-323.

[5] Chen T Y, Poon P L, Xie X. METRIC: METamorphic Relation Identification based on the Category-choice framework [J]. Journal of Systems & Software, 2015, 116:177-190.

[6] Chen T Y, Poon P L, Tse T H. A Choice Relation Framework for Supporting Category-Partition Test Case Generation [J]. IEEE Transactions on Software Engineering, 2003, 29(7):577-593.

[7] Liu H, Poon P L, Chen T Y. Enhancing partition testing through output variation [C]//Proceedings of the 37th International Conference on Software Engineering. IEEE, 2015: 805-806.

[8] Sun C, Liu Y, Wang Z, et al. μMT: A Data Mutation Directed Metamorphic Relation Acquisition Methodology[C]//IEEE/ACM International Workshop on Metamorphic Testing. IEEE, 2017:12-18.

[9] Ma Y S. MuJava: a mutation system for java[C]// Proceedings of the 28th International Conference on Software Engineering (ICSE2006). ACM, 2006:827-830.

[10] Xie X, Li J, Wang C, et al. Looking for an MR? Try METWiki today[C]// In proceedings of the 1st International Workshop on Metamorphic Testing. IEEE, 2017:1-4.

[11] Kanewala U, Bieman J M. Using machine learning techniques to detect metamorphic relations for programs without test oracles[C]// In proceedings of the 24th International Symposium on Software Reliability Engineering(ISSRE’13). IEEE, 2014:1-10.

[12] Liu H, Liu X, Chen T Y. A New Method for Constructing Metamorphic Relations[C]// In proceedings of the 12th International Conference on Quality Software (QSIC’12). IEEE, 2012:59-68.