Contents lists available at ScienceDirect

Information and Software Technology

journal homepage: www.elsevier.com/locate/infsof

Validating class integration test order generation systems with Metamorphic Testing

Miao Zhang [a,*], Jacky Wai Keung [a], Tsong Yueh Chen [b], Yan Xiao [c]

[a] *Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, China*
[b] *Department of Computer Science and Software Engineering, Swinburne University of Technology, Hawthorn, VIC 3122, Australia*
[c] *School of Computing, National University of Singapore, 117417, Singapore*

ARTICLE INFO

Keywords:
Integration testing
Class integration test order generation
Metamorphic Testing
Metamorphic Relations

ABSTRACT

Context: Previous studies proposed different kinds of approaches for class integration test order generation, and corresponding systems can be implemented based on these approaches. Such class integration test order generation systems can facilitate the process of software integration testing if they are implemented correctly.
Objective: However, a test oracle problem exists in the class integration test order generation systems. Since these approaches for class integration test order generation normally deliver a local optimum rather than a global optimum, there are no practically feasible ways to validate their generated class integration test orders, that is, these implementation systems are untestable.
Method: To address the test oracle problem, we apply Metamorphic Testing (MT) to validate class integration test order generation systems. Metamorphic Relations (MRs), which are the key components of MT, reason about relations between test outputs of a system. Five effective MRs are developed to ensure the quality of the class integration test order generation systems. In these proposed MRs, follow-up test inputs are generated by modifying classes or class dependencies in the source test inputs while some characteristics of the source test outputs are preserved, for example, the same class integration test order or the equal stubbing cost. Faults can be detected in systems if an individual MR is violated for certain tests.
Results: Failure detection of MT has been successfully demonstrated in empirical experiments on three systems implementing different typical class integration test order generation approaches. More than 84% of faulty programs can be detected by all MRs, for three class integration test order generation systems under investigation.
Conclusion: The experimental results show that the proposed MRs are able to systematically and effectively detect faults in class integration test order generation systems. This study explores a new application domain in MT and further extends its applications in Software Engineering.

1. Introduction

Software integration testing is an essential process to guarantee the quality of software being produced. An important problem in software integration testing is to determine the order in which classes are integrated and tested, because classes in a software system are usually independently developed and tested. Test stubs are constructed for a class *A* when classes on which class *A* depends are unavailable. Because cycles among class dependencies are common in programs [1], test stubs are inevitable.

Different class integration test orders (CITOs) need various test stubs, and correspondingly, require different efforts to construct such test stubs, which has been proved by Kung et al. [2]. In their experiment, students were asked to prepare test stubs for some member functions for a program, and one simple test stub is required about 0.79 person-hours to prepare. For a case study system called InterViews which contains 122 classes, a random CITO requires 191 test stubs that could be written in about 152 person-hours, while the optimal CITO requires only 8 test stubs that can be constructed in around 7 person-hours. Therefore, for an ideal CITO, few or no efforts required to construct test stubs for the integration testing are desirable.

Finding the optimal CITO is basically equivalent to the traveling salesman problem [3], which is well-known as a factorial ordered problem. Searching for the globally optimal CITO is impractical for a

[*] Corresponding author.
E-mail addresses: miazhang9-c@my.cityu.edu.hk (M. Zhang), jacky.Keung@cityu.edu.hk (J.W. Keung), tychen@swin.edu.au (T.Y. Chen), dcsxan@nus.edu.sg (Y. Xiao).

https://doi.org/10.1016/j.infsof.2020.106507
Received 14 May 2020; Received in revised form 28 September 2020; Accepted 7 December 2020
Available online 16 December 2020

real-life application even with a moderately large number of classes, and hence heuristics are extensively adopted to search for the locally optimal CITO instead.

Previous studies focus on the improvement of the performance for different kinds of class integration test order generation approaches. Systems can be implemented based on these proposed approaches to generate class integration test orders for a given Object-Oriented program. Systems implemented based on such approaches can facilitate the process of software integration testing. However, no matter how good the approach for class integration test order generation is, an incorrect implementation would not benefit from its capability. Therefore, it is important to validate such class integration test order generation systems.

However, testing such a system is difficult if not impossible, because except for trivial inputs, no practically feasible ways exist to validate its generated class integration test order. Approaches for class integration test order generation would be more likely to deliver a locally optimal solution than a globally optimal solution. Given a real-life system of even moderate complexity, we are not able to validate whether a generated class test order for this real-life system is the actual output that should be computed by the approach, as we are unable to give an exact definition for the concept of local optimum. Overall, such class test order generation systems are referred to as untestable systems or systems with the test oracle problem.

Fortunately, the technique of metamorphic testing (MT) [4] could help to alleviate the test oracle problem. MT differs from conventional testing approaches which focus on the verification of each individual output of the software under testing, instead MT verifies relations among the inputs and outputs of multiple executions of the software. Such relations are referred to as metamorphic relations or MRs for short. If an individual MR is violated for certain tests, it implies that the implementation is faulty because MRs are the necessary properties of the implemented algorithm. In this case, the test oracle is not required, which is suitable for our untestable systems.

We propose to use MT as a framework for validating the class integration test order generation systems. MT has never been applied in the area of class integration test order generation, since no references to the applicable MRs which are the key components in MT can be obtained. A challenging problem is to develop effective MRs for validating these systems, which is one of the major contributions delivered by this paper.

In this paper, we propose five kinds of MRs to validate different CITO generation systems. These MRs modify class dependencies with the aim of preserving a part of or the entire original test order. We validate three systems implementing different typical CITO generation approaches using the test suites generated based on the proposed MRs. The results show that no faults can be detected by the five proposed MRs in three CITO generation systems, which increases the likelihood of being valid for our systems. We also conduct mutation analysis to evaluate the effectiveness of the proposed MRs. The experimental results show that more than 84% of faulty programs can be detected by all MRs for three class integration test order generation systems. The contributions of this paper are as follows:

- We validate three systems implementing different class integration test order generation approaches based on MT. To our best knowledge, this is the first time to introduce MT in class integration test order generation.
- Five kinds of MRs are proposed to test the class integration test order generation systems effectively, which can be extended into all systems to generate class integration test orders.
- Experiments are conducted on three systems to demonstrate the failure detection capability of MT.

The remainder of this paper is organized as follows. Section 2 introduces the background, including class integration test order generation issues and MT. We present our validation framework and the proposed MRs in Section 3. The experiments follow in Section 4. Section 5 concludes this work.

2. Background

This section will briefly introduce the class integration test order generation problem, the existing approaches, and the preliminary knowledge of MT.

2.1. Class integration test order generation issue

Given a class integration test order, test stubs are constructed for a class to provide services from the classes on which it depends by emulating the behaviors of these classes. Suppose that class C_i depends on class C_j, if class C_i is integrated and tested prior to class C_j based on the generated test order, a test stub is created for class C_i to provide services from class C_j. To estimate the cost to construct such a test stub $Stub(C_i, C_j)$, Briand et al. [3] proposed a metric, stubbing complexity (SCplx), which is calculated as follows:

$$SCplx(i,j) = [W_A \cdot \overline{A(i,j)}^2 + W_M \cdot \overline{M(i,j)}^2]^{1/2} \tag{1}$$

In this equation, attributing complexity $A(i,j)$ and method complexity $M(i,j)$ counts the number of attribute assess and method invocation from class C_i to class C_j, respectively. These two kinds of complexity are equally important in affecting the cost to construct such a test stub, therefore, two weights for two kinds of complexity are equal, i.e., $W_A = W_M = 0.5$. Correspondingly, to estimate the cost to construct all test stubs based on a given class test order O, a metric, overall stubbing complexity (OCplx) is proposed in [3]. It is calculated as the sum of SCplx of all test stubs, which is shown as (2).

$$OCplx(O) = \sum_{(i,j)\in Stubs} SCplx(i,j) \tag{2}$$

If no cycles among class dependencies exist in the program under integration testing, a class integration test order can be generated by the reverse topological sorting of all classes, and no test stubs are required. However, constructing test stubs are unavoidable when circular dependencies exist among classes in the program, which is a common phenomenon. Existing studies aimed at devising a class test order with lower overall stubbing complexity and proposed different kinds of approaches for this issue.

In this paper, we focus on one popular kind of class integration test order generation approaches, which is based on graph theory. Given a program under integration testing, this kind of approaches first describe class dependencies in this program through constructing an object relation diagram (ORD). In such a diagram, each node represents a class and each directed edge represents a class dependency from one class (source node) to the target class (target node) on which it depends. Fig. 1 shows an ORD for a sample program containing six classes. Values above the edge indicate the corresponding SCplx for each edge. For example, $C_1 \rightarrow C_2$ indicates class C_1 depends on class C_2, and the SCplx is 0.3. Four cycles exist in this ORD:

(1) $C_1 \rightarrow C_5 \rightarrow C_6 \rightarrow C_1$
(2) $C_1 \rightarrow C_5 \rightarrow C_4 \rightarrow C_1$
(3) $C_1 \rightarrow C_3 \rightarrow C_4 \rightarrow C_1$
(4) $C_1 \rightarrow C_2 \rightarrow C_3 \rightarrow C_4 \rightarrow C_1$

To generate the final class integration test order, such cycles must be broken by removing some edges and test stubs are constructed for the corresponding class dependencies. The OCplx of such class integration test order can be calculated based on the constructed test stubs. Therefore, the adopted cycle-breaking algorithms are important, which affect the removed edges and the final stubbing cost.

Most existing cycle-breaking algorithms are greedy algorithms: they assign weights for different edges to assess the stubbing cost for the corresponding class dependencies and remove the edge with the minimal or maximal weight that indicates the lowest stubbing cost in each step. Such operation will be repeated until no cycles exist. These cycle-breaking algorithms can be characterized as three types based on their weight calculation.

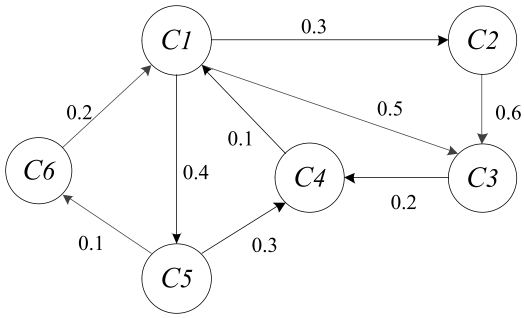


Fig. 1. Object relation diagram for a sample program. The sample program contains six classes. Each node represents a class. Each directed edge from source node to target node represents the source class depends on the target class, and the value above the edge represents the corresponding SCplx for each edge.

(1) *Number of cycles.* The weight for each edge is calculated as the number of cycles in which the edge is involved. For example, Briand et al. [5] roughly estimated the number of cycles in which an edge is involved by the product of in-degree of source node and out-degree of target node. The edge that is involved in more cycles is desired because removing such an edge can break more cycles at once. For the sample program, the edge $C_4 \rightarrow C_1$ will be removed first, which breaks three out of four cycles.

(2) *Stubbing cost.* The stubbing cost for each class dependency is calculated as the weight for the corresponding edge, such as the SCplx calculated by Eq. (1). Hashim [6] measured the effort to construct test stubs for each class as its weight, the class with the minimal weight is identified and test stubs are created to emulate the classes on which it depends. For the sample program, the edge $C_4 \rightarrow C_1$ and $C_5 \rightarrow C_6$ with the lowest SCplx will be removed first.

(3) *Cycles-weight ratio.* The weight for each edge is calculated combining the abovementioned two factors, which is the ratio of the number of cycles and stubbing cost. Bansal et al. [7] and Abdurazik et al. [8] used such an approach. Removing the edge with the highest value means that we can break more cycles at a relatively lower stubbing cost. For the sample program, the edge $C_4 \rightarrow C_1$ having the highest cycles-weight ratio will be removed first.

Although these cycle-breaking algorithms adopted different methods for weight calculation, the procedure to generate class integration test orders is identical. Systems implementing these cycle-breaking algorithms are similar except for weight calculation.

2.2. Metamorphic testing (MT)

Metamorphic Testing (MT) is first proposed by Chen et al. [4,9] to alleviate the problem of test oracles by verifying the relations among the inputs and outputs of multiple executions for the programs or systems under testing. Metamorphic Relation (MR) is the key component in MT, which reasons about relations between test outputs of a program when it lacks test oracles. In general, a MR is a necessary property of a function f over function inputs $x_1, x_2, \ldots, x_n$ $(n > 1)$, and their corresponding output values $f(x_1), f(x_2), \ldots, f(x_n)$ [10,11]. For example, a mathematical property of the *sine* function is that $\sin(x) = \sin(\pi - x)$, the corresponding MR is that if two function inputs x_1 and x_2 satisfy $x_1 + x_2 = \pi$, then two function outputs should be equal, i.e., $\sin(x_1) = \sin(x_2)$. In this example, x_2 is constructed based on $\langle x_1, \sin(x_1)\rangle$ according to the MR, therefore, we refer to x_1 as a source input and x_2 as a follow-up input. Such relations can be used to test the program when we are not sure whether the test output is correct. An implementation for this function must have faults if it violates the above relation.

Many studies from different domains have proved the effectiveness of MT to alleviate the test oracle problem, such as, epidemiological [12,13], big data analytics software [14,15], and security-critical applications (cybersecurity [16] and cryptography [17,18]). Especially in compilers [19,20], MT has helped to detect many real-life faults. Le et al. [19] used a very simple MR and detected 147 unique bugs in two popular C compilers (GCC and LLWM) in 2014. But they are not the only researchers to use MT to test compilers. In 2010, Tao et al. [20] also conducted the testing of similar compilers, but they only found one fault in the GCC compiler. The number of faults detected by two research teams is dramatically different, which means MRs have a significant impact on the MT's capability for fault detection. Furthermore, as observed by Liu et al. [21], a small number of diverse MRs may effectively reveal almost all faults. Therefore, we aim to analyze different kinds of MRs, develop more effective MRs, and demonstrate the practicality and robustness of class integration test order generation systems through a comprehensive scaled experiment.

3. Our validation framework

3.1. Validation procedure

In this paper, we adopt MT to validate systems implementing the class integration test order generation approaches introduced in Section 2. MR is the key component in MT, which is a necessary property of the function over its inputs and the corresponding outputs. MR describes how to generate a follow-up test case based on the source test case and its source output, and explains which relation between two outputs (i.e., source output and follow-up output) must be satisfied. We first extract MRs for these systems implementing the class integration test order generation approaches. The proposed MRs are generated based on the necessary property of these approaches, which will be explained in detail in Section 3.2. The main procedure of these class integration test order generation approaches is identical, which is to break all cycles in the object relation diagram constructed for the program under integration testing. Once no cycle exists in the diagram, classes are integrated and tested based on the reverse topological sorting. Test stubs are only constructed for the class dependencies whose corresponding edges are removed in cycles.

Our validation procedure for the class integration test order generation systems includes four steps. Firstly, we select source programs as source inputs to run the class integration test order generation system and obtain the generated test orders as source outputs. Then, follow-up programs are generated based on the proposed MR, source inputs and their corresponding source outputs. Such follow-up programs are referred to as follow-up inputs. Next, follow-up inputs are used to run the class integration test order generation system. The obtained test orders are follow-up outputs. Finally, we check whether the proposed MR holds for such two kinds of outputs. If the relation is violated, a failure is revealed.

3.2. Metamorphic relations (MRs)

In this section, we discuss the developed MRs for the class integration test order generation systems.

(1) **MR1: Permutation of class names**

An obvious MR to validate class integration test order generation systems is about permuting class names. Fig. 2 shows an example for the permutation of class names. In the source program and follow-up program, each node represents a class and each directed edge represents a class dependency, for example, $C_1 \rightarrow C_2$ represents class C_1 depends on class C_2. Let *SP* denote the source program that contains *six* classes, and the output is a class integration test order, O_{sp}, which is further supposed to be $[C_4, C_5, C_3, C_2, C_1, C_6]$. A follow-up program *FP* can be constructed by permuting the class names without any modifications on inter-class relationships and functionality of these

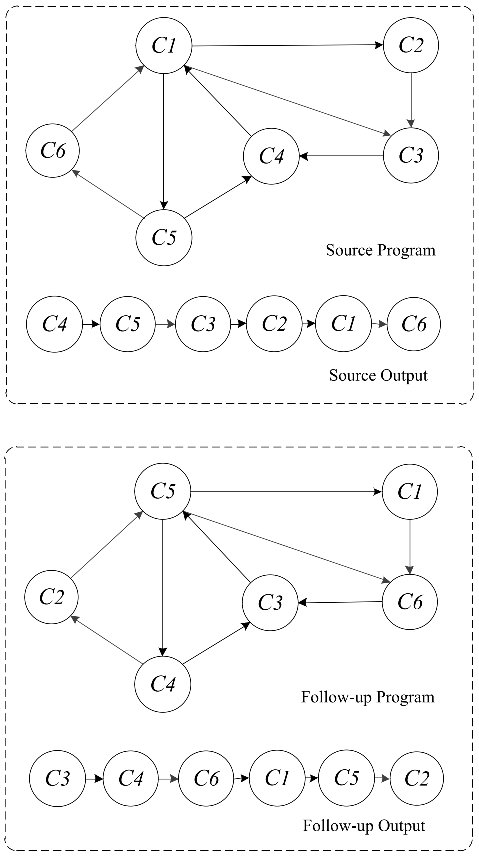


Fig. 2. Example of MR1 (Permutation of class names). The adopted permutation scheme is $C_1 \rightarrow C_5$, $C_2 \rightarrow C_1$, $C_3 \rightarrow C_6$, $C_4 \rightarrow C_3$, $C_5 \rightarrow C_4$, $C_6 \rightarrow C_2$.

classes. Obviously, there is a one-to-one correspondence between the computed test orders of programs *SP* and *FP*, and the efforts required to construct test stubs based on these two test orders should also be equal.

(2) **MR2: Adding/deleting class dependencies for the initial class with the aim of preserving its test order**

Fig. 3 shows an example for MR2. The notation for nodes and edges is similar with Fig. 2, and the weight for each edge represents the stubbing cost for each class dependency. Let *SP* denote the source program that contains *six* classes, and in the output, i.e., the generated class integration test order, O_{sp}, class C_4 is integrated and tested first. It indicates that the highest test priority of class C_4 will contribute to a lower stubbing cost for the subsequent integration testing (i.e., test stubs are not required for those classes depending on class C_4). In such a case, adding new class dependencies to class C_4 will not incur extra stubbing cost for the subsequent integration testing, and this operation will not affect the test order of class C_4. Hence, a follow-up program *FP* which has the same classes as *SP*, can be generated by adding one or more class dependencies to class C_4, for example, the addition of edge $C_2 \rightarrow C_4$. Class C_4 will still be the initial class in the test order O_{fp} generated for the program *FP*. Because the classes on which class C_4 depends are unchanged, the stubbing cost of O_{fp} will be equal to that of O_{sp}. Similarly, deleting class dependencies from class C_4, will reduce the stubbing cost when class C_4 is integrated, and as a consequence, this operation will not affect its test priority. Therefore, the other follow-up program *FP* which has the same classes as *SP*, can be constructed by removing class dependencies through the removal of edges, such as, removing class C_1 on which class C_4 depends, through the removal of edge $C_4 \rightarrow C_1$. Class C_4 will still be the initial class in the class integration test order O_{fp} generated for the program *FP*. Because the classes on which class C_4 depends are fewer in *FP*, the stubbing cost of O_{fp} will be smaller than that of O_{sp}.

To prove MR2, we need the following proposition:

Proposition 1. *Suppose class C is integrated and tested first. If class C is not independent, all class dependencies from class C will be involved in cycles.*

Proof of Proposition 1. Suppose that there exists another class D, class C depends on class D and the class dependency $C \rightarrow D$ is not involved in any cycles. According to our assumption that all systems for class integration test order generation will not construct unnecessary test stubs when no cycle exists, class D should be integrated and tested prior to class C, so that a test stub created for class C to emulate class D can be avoided. Although class D might be involved in cycles, we can regard class D and other classes that are involved in the same cycles as a whole. The test order of other classes can be ignored because it will not affect the test sequence between class D and class C. Class C will not be the initial class in the class integration test order, which contradicts our assumption that class C is integrated and tested first. Therefore, the class dependency $C \rightarrow D$ must be involved in cycles.

Proof of MR2. Class C is integrated and tested first, which means all class dependencies from class C are removed and test stubs are constructed for these class dependencies. Because all class dependencies from class C are involved in cycles, such class dependencies must be the locally optimal solution obtained by approaches to break cycles in which they are involved. To generate the follow-up program, we remove some of these class dependencies from class C, the corresponding cycles will be broken automatically. This operation will not affect remaining cycles and solutions to break these cycles. Remaining class dependencies from class C will still be the locally optimal solution for the follow-up program, and corresponding test stubs are constructed. Therefore, class C will still be the initial class in the test order.

Another measure to generate the follow-up program is to add new class dependencies to class C. If such class dependencies do not form new cycles, the locally optimal solution obtained by class integration test order generation approaches will be the same, and class C will still be the initial class in the integration test order. If the newly added class dependencies form new cycles, some class dependencies from class C must be involved in these cycles. Removing such class dependencies from class C will be the optimal solution to break these cycles if they own lower stubbing cost. Therefore, to make sure that class C will still be the initial class in the class integration test order, the newly added class dependencies should be assigned a very large weight W.

(3) **MR3: Adding/deleting class dependencies for the last class with the aim of preserving its test order**

Fig. 4 shows an example for MR3, where the notation for nodes and edges is similar with that of other MRs. Let *SP* denote the source program that contains *six* classes, and in its generated class integration test order, O_{sp}, class C_6 is the last class to be integrated and tested. That means testing class C_6 requires the greatest stubbing cost, or this operation cannot lower stubbing cost for the subsequent integration testing (i.e., fewer classes depending on class C_6). A MR can be defined relating to the modification of class dependencies for the last class in the integration test order, i.e., class C_6. Because testing class C_6 has already incurred the greatest stubbing cost, the increase of its stubbing cost by adding more class dependencies from class C_6 will not affect the test order of class C_6. Hence, a follow-up program *FP* which has the same classes as *SP*, can be produced by adding one or more dependencies from class C_6 to other classes, for example, adding edge $C_6 \rightarrow C_2$. Class C_6 will still be the last class in the class integration test order O_{fp} generated for the program *FP*. Because class C_2 is integrated and tested prior to class C_6, no extra test stubs are required in the

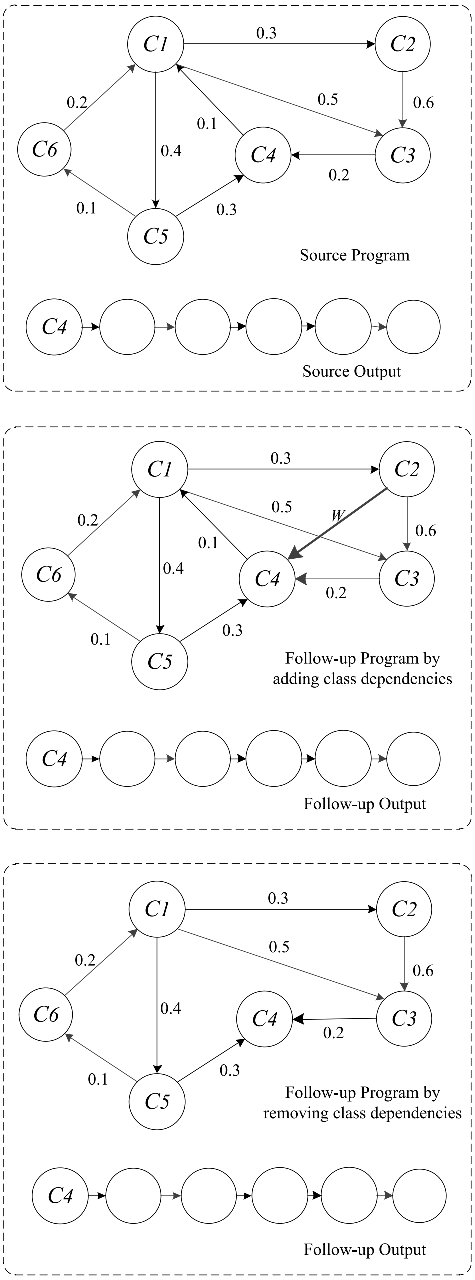


Fig. 3. Example of MR2 (Adding/deleting class dependencies for the initial class). The added class dependency is $C_2 \rightarrow C_4$ with a very large weight W, and the removed class dependency is $C_4 \rightarrow C_1$.

program *FP*. The stubbing cost of O_{fp} will be the same as that of O_{sp}. Similarly, reducing the stubbing cost saved by the integration of class C_6 for the subsequent integration testing has no effects on the test order of class C_6. Therefore, suppose that class C_6 is depended by some other classes, a follow-up program *FP* which has the same classes as *SP*, can be generated by removing one or more class dependencies to class C_6, such as, removing edge $C_5 \rightarrow C_6$. Class C_6 will still be the last class in the test order O_{fp} generated for the program *FP*. The test stub constructed for class C_5 to emulate the services from class C_6 is unnecessary now. The stubbing cost of O_{fp} will be lower than that of O_{sp}.

To prove MR3, we need the following proposition:

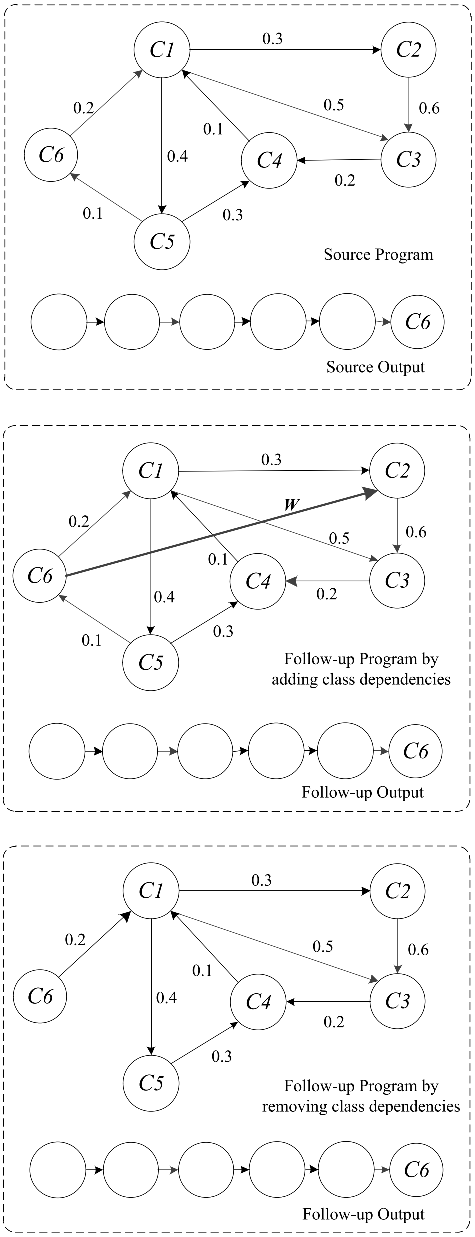


Fig. 4. Example of MR3 (Adding/deleting class dependencies for the last class). The added class dependency is $C_6 \rightarrow C_2$ with a very large weight W, and the removed class dependency is $C_5 \rightarrow C_6$.

Proposition 2. *Suppose that class C is the last class to be integrated and tested. If other classes depend on class C, all class dependencies to class C are involved in cycles.*

Proof of Proposition 2. Suppose that there exists another class B, such that class B depends on class C and the class dependency $B \rightarrow C$ is not involved in any cycles. Class C should be integrated and tested prior to class B, so that a test stub created for class B to emulate class C can be avoided according to our assumption that unnecessary test stubs are not constructed. Then, class C will not be the last class in the integration test order, which contradicts our assumption. Therefore, the class dependency $B \rightarrow C$ must be involved in cycles.

Proof of MR3. Class C is integrated and tested lastly, which means all class dependencies to class C are the locally optimal solution obtained by class integration test order generation approaches to break cycles in

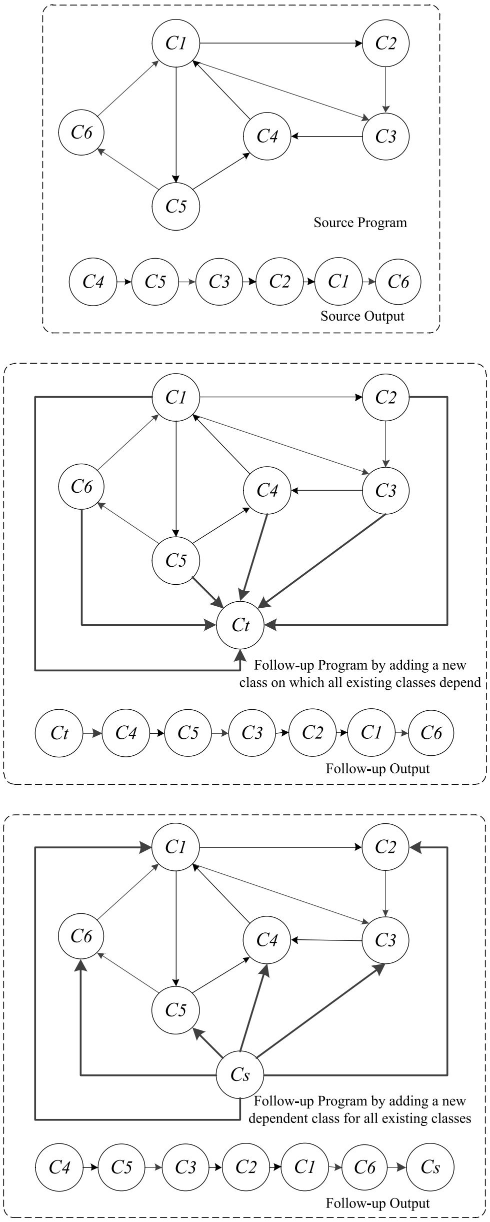

C1
C2
C6
C4
C3
C5
Source Program
C4 C5 C3 C2 C1 C6
Source Output
Ct
Follow-up Program by adding a new
class on which all existing classes depend
Ct C4 C5 C3 C2 C1 C6
Follow-up Output
Cs
Follow-up Program by adding a new
dependent class for all existing classes
C4 C5 C3 C2 C1 C6 Cs
Follow-up Output

Fig. 5. Example of MR4 (Adding a new class with the aim of preserving the original test order). One kind of added class is C_t, which is depended by all the existing classes, and the other kind is C_s, which depends on all other existing classes.

which they are involved. Removing some class dependencies to class C breaks their corresponding cycles and does not affect the solutions to break the remaining cycles. Therefore, test stubs will be constructed for other remaining class dependencies to class C, and class C will still be the last class in the class integration test order.

For adding new class dependencies from class C, if such class dependencies do not form new cycles, the classes on which class C depends will be integrated and tested before class C, which is consistent with the fact that class C is integrated and tested lastly. If the newly added class dependencies form new cycles, some existing class dependencies to class C must be also involved in these cycles. To make sure that removing these class dependencies to class C will also be the optimal solution for the new cycles, a very large weight W should be assigned to the newly added class dependencies.

(4) **MR4: Adding a new class with the aim of preserving the original test order**

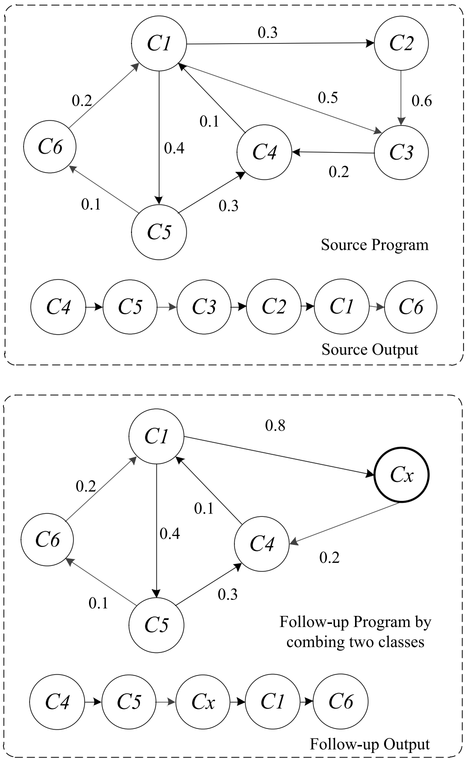


Fig. 6. Example of MR5 (Combining classes while preserving the original test order). Classes C_2 and C_3 are combined as a new class C_x.

Different from MR2 and MR3 in which classes in the program remain unchanged, a new class is added to the program with the aim of preserving the original test order in MR4. Fig. 5 shows an example for MR4. Let *SP* denote the source program that contains *six* classes, and the class integration test order generated for the program *SP* is O_{sp}, which is further supposed to be $[C_4, C_5, C_3, C_2, C_1, C_6]$. A follow-up program *FP* can be generated by adding a new class C_t such that all the classes in the program depend on this new class C_t. Therefore, the class integration test order O_{fp} generated for the program *FP* should be $O_{fp} = [C_t, O_{sp}]$. Similarly, a new class C_s that depends on all other existing classes is added, and the class integration test order O_{fp} should be $[O_{sp}, C_s]$. The stubbing cost of these two class integration test orders O_{fp} will be the same as that of O_{sp}.

(5) **MR5: Combining classes with the aim of preserving the original test order**

A more complicated MR is to combine some classes with the aim of preserving the original test order. Fig. 6 shows an example for MR5. Let *SP* denote the source program that contains *six* classes, and the class integration test order generated for the program *SP* is O_{sp}, which is further supposed to be $[C_4, C_5, C_3, C_2, C_1, C_6]$. Two edges $C_4 \rightarrow C_1$ and $C_5 \rightarrow C_6$ are removed to break cycles, and test stubs are constructed for the two corresponding class dependencies. A follow-up program *FP* can be generated by combining classes, for example, combining classes C_2 and C_3 to form a new class C_x. Dependencies between class C_2 and C_3 are ignored, while dependencies between class C_2 (or C_3) and other remaining classes are generated for the new class C_x. For instance, an edge $C_1 \rightarrow C_x$ with the weight as 0.8 replaces the edge $C_1 \rightarrow C_2$ (weight = 0.3) and the edge $C_1 \rightarrow C_3$ (weight = 0.5). In the follow-up program, the combination of classes C_2 and C_3 does not affect the class integration test order generated by approaches, therefore, the class integration test order generated for the program *FP* is similar with O_{sp}, that is, $O_{fp} = [C_4, C_5, C_x, C_1, C_6]$. The stubbing cost of O_{fp} will be the same as that of O_{sp}.

To preserve the original test order, the combination of appropriate classes is very important. As we have known the generated class integration test order for the source program, we can easily infer which edges are removed to break cycles. Suppose that an edge e is removed to break m ($m > 1$) cycles, and c is one of these cycles. If a class dependency $C_i \rightarrow C_j$ is only involved in this cycle c, combining C_i and C_j will not affect the generated class test order, because the class dependency $C_i \rightarrow C_j$ is not the optimal solution to break cycles, i.e., the edge e will still be removed to break such m cycles.

Based on the above five MRs, follow-up inputs are generated. Then follow-up outputs can be obtained by running the class integration test order generation system. Finally, we check whether follow-up outputs and source outputs satisfy the corresponding MR. Faults are detected if the MR is violated.

4. Experiments

This section provides an experimental evaluation of our validation framework for three class integration test order (CITO) generation systems. Test data, including source test cases and follow-up test cases are generated based on the proposed MRs. Firstly, source test cases and follow-up test cases are applied in three original CITO generation systems to detect any violations of these proposed MRs. If any violation is detected, which means that faults exist, then we will modify the systems to correct these faults. After that, mutation analysis is adopted to evaluate the effectiveness of five proposed MRs. Mutants are generated for the CITO generation system as faulty programs. If an MR is violated, a mutant is killed and correspondingly, a fault can be detected. Section 4.1 introduces the information of three CITO generation systems. Section 4.2 introduces the experimental design. Generation of test cases and mutants are described in Sections 4.3 and 4.4, respectively. Section 4.5 provides the experimental results and analyses. Section 4.6 presents the discussion, and Section 4.7 discusses the threats to validity.

4.1. CITO generation systems

Fig. 7 shows the overview of three CITO generation systems. These systems are implemented by us based on the corresponding CITO generation approaches introduced in Section 2, because there are no public CITO generation systems. Given a program under the integration testing, CITO generation system takes the information of classes and class dependencies as the input and generates a class order for integration testing as an output. For each class, class information presents its dependent classes, and the classes on which it depends. Class dependency information indicates attribute coupling and method coupling for each class dependency, which can be used to calculate its stubbing cost. Based on such class and class dependency information, an object relation diagram is constructed for the given program and cycles are identified in the diagram. We assign different weights to each edge measuring the effort of removing it. The appropriate edge is removed to break cycles based on its weight. After all cycles have been broken, a class integration test order is generated. Three CITO generation systems are implemented based on different kinds of weight calculation:

- NCSystem: weight is calculated based on the number of cycles. The edge that is involved in the greatest number of cycles will be removed.
- SCSystem: weight is calculated based on stubbing cost. The edge that requires the lowest stubbing cost will be removed.
- CWRSystem: weight is calculated based on cycles-weight ratio. The edge with the highest cycles-weight ratio is removed.

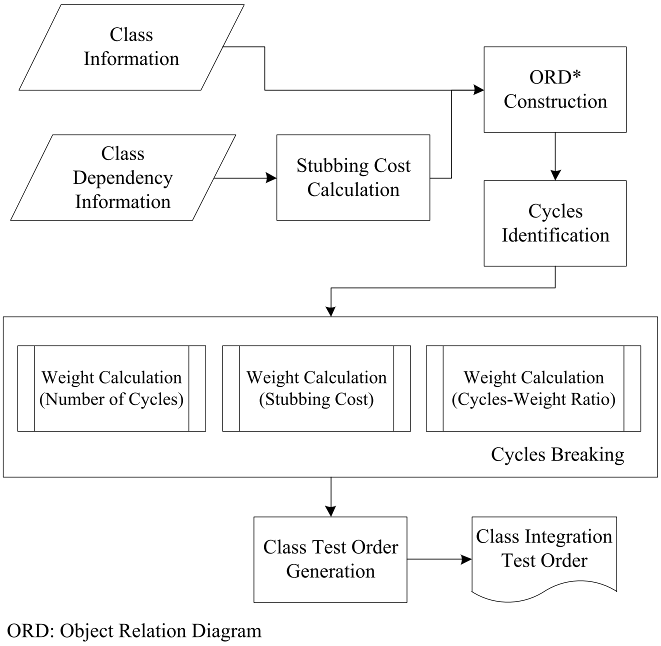

Class Information
Class Dependency Information
Stubbing Cost Calculation
ORD* Construction
Cycles Identification
Weight Calculation (Number of Cycles)
Weight Calculation (Stubbing Cost)
Weight Calculation (Cycles-Weight Ratio)
Cycles Breaking
Class Test Order Generation
Class Integration Test Order
ORD: Object Relation Diagram

Fig. 7. Overview of class integration test order generation systems.

Table 1
Information of source programs.

Programs	Classes	Deps	Cycles	Deps in Cycles	LOC
ANT	25	83	654	41	4093
ATM	21	67	30	29	1390
DNS	61	276	16	23	6710
SPM	19	72	1178	59	1198

Multiple edges may satisfy the above criteria for the cycle-breaking in practice. In case of a tie, we propose the following measures for this issue. If the weight calculation is based on the number of cycles, the edge attracting the lowest stubbing cost is selected from its competitors. If the weight calculation is based on stubbing cost or cycles-weight ratio, the edge involved in the greatest number of cycles is selected. Otherwise, the edge where the source node has the smallest index is selected. Similarly, multiple classes can be integrated at the same time if all classes on which they depend have been tested. Hence, such multiple classes have the same test order.

4.2. Experimental design

We first generate test suites based on the proposed MRs. Then we test three CITO generation systems using the given test suites. Once a MR is violated, faults can be detected. We can modify our systems based on the violated MR and the corresponding test suites until no faults can be found. We conduct experiments to answer the following two questions.

RQ1: What is the performance of different MRs on detecting faulty programs? As we have known, different MRs have different impacts on the MT's capability for fault detection. We evaluate the effectiveness of different MRs on detecting faults. Mutants are generated as faulty programs. A mutant is killed if the given MR is violated, and correspondingly, faults are detected.

RQ2: What is the performance of MT on detecting different kinds of faults? Because these faults generated by mutation operators are very common in practice. We aim to validate the capability of MT in detecting different kinds of faults. Therefore, we count the number of killed mutants generated by different mutation operators. By comparing the number of different kinds of killed mutants, we can find for which

Table 2
Number of follow-up programs for MRs. For MR2 and MR3, column ADD (DEL) counts the number of follow-up programs generated by adding (deleting) class dependencies for the initial or last class. For MR4, column SRC (TAR) counts the number of follow-up programs generated by adding a source (target) class.

SUT	Programs	MR1	MR2		MR3		MR4		MR5	SUM
			ADD	DEL	ADD	DEL	SRC	TAR		
NCSystem	ANT	5	5	0	5	0	5	5	1	26
	ATM	5	5	0	5	1	5	5	2	28
	DNS	5	5	0	5	2	5	5	4	31
	SPM	5	5	3	5	1	5	5	3	32
	SUM	20	20	3	20	4	20	20	10	117
SCSystem	ANT	5	5	1	5	0	5	5	1	27
	ATM	5	5	0	5	1	5	5	2	28
	DNS	5	5	0	5	2	5	5	4	31
	SPM	5	5	3	5	1	5	5	1	30
	SUM	20	20	4	20	4	20	20	8	116
CWRSystem	ANT	5	5	1	5	0	5	5	0	26
	ATM	5	5	0	5	1	5	5	2	28
	DNS	5	5	0	5	2	5	5	4	31
	SPM	5	5	3	5	1	5	5	3	32
	SUM	20	20	4	20	4	20	20	9	117

kinds of faults the proposed MRs performed well, and for which kinds of faults the proposed MRs performed the worst, which can provide the direction for our future work, i.e., to propose more specific MRs for those kinds of faults that are difficult to detect now.

4.3. *Test case generation*

For the source test cases, we choose four benchmark programs that are popularly used in the class integration test order generation problem as source inputs. Table 1 presents the information of source programs, i.e., these benchmark programs, including the number of classes, class dependencies (Deps), cycles, dependencies that are involved in cycles, and LOC. The detailed information of classes and class dependencies, such as attribute coupling and method coupling, can be found in the literature [3].

We run three CITO generation systems for these benchmark programs and obtain the corresponding class integration test orders as the source outputs. Follow-up test cases can be generated easily based on source inputs, source outputs, as well as these proposed MRs. For some MRs, more than one follow-up program can be generated according to a source test case, therefore, we randomly choose five follow-up programs in such cases. Table 2 counts the number of follow-up programs for each MR.

As shown in Table 2, total 117, 116 and 117 follow-up programs are generated for NCSystem, SCSystem and CWRSystem, respectively. For MR1, i.e., the permutation of class name, which is the simplest, five follow-up test cases are generated for each source test case. For MR2 and MR3, adding class dependencies for the initial (last) class is much easier than deleting class dependencies for the initial (last) class with the aim of preserving its test order. In some cases, such as ANT for NCSystem, the initial class in the class test order does not depend on any other classes. Similarly, no other classes depend on the last class in the test order. Therefore, no class dependencies from the initial class (or to the last class) can be removed. For MR4, adding a target class on which all the existing classes depend, or a source class which depends on all the existing classes, is easy to implement. Therefore, 20 follow-up test cases are generated for both situations respectively. For MR5, only certain class combinations may preserve the class test order and remain the same stubbing cost. Therefore, it is more complicated to generate such follow-up test cases compared with other MRs.

4.4. *Mutant generation*

To evaluate the effectiveness of the proposed MRs, we generate mutants for three CITO generation systems by the tool, PIT.[1] PIT is a mutation testing system, which runs the provided test cases against mutants, i.e., faulty programs. PIT first compiles the given source code to generate byte code. Then faulty programs are obtained by applying a set of mutation operators to the byte code. Different results are produced when the application code changes, and correspondingly, such faulty programs can be detected. Mutation operators used in PIT are described as the following:

- **FALSE_RETURNS** (False Returns Mutator): Replace primitive and boxed Boolean return values with false.
- **CONDITIONALS_BOUNDARY** (Conditionals Boundary Mutator): Replace the relational operators, such as, <, >, with their boundary counterpart, i.e., <=, >=.
- **EMPTY_RETURNS** (Empty Returns Mutator): Replace return values with an 'empty' value for that type. For example, replacing *java.util.List* with *Collections.emptyList*().
- **INCREMENTS** (Increments Mutator): Replace increments with decrements and vice versa. For example, change increment from 1 to −1 and change increment from −1 to 1.
- **VOID_METHOD_CALLS** (Void Method Call Mutator): Remove method calls to void methods.
- **MATH** (Math Mutator): Replace binary arithmetic operations for integer or floating-point arithmetic with another operation. For example, replace integer subtraction with addition, replace double addition with subtraction, replace double division with multiplication, and replace double multiplication with division.
- **NEGATE_CONDITIONALS** (Negate Conditionals Mutator): Replace the original conditional with its negate conditional.
- **NULL_RETURNS** (Null Returns Mutator): Replace return values with null.[2]
- **PRIMITIVE_RETURNS** (Primitive Returns Mutator): Replace int, short, long, char, float and double return values with 0.

Table 3 counts the number of different kinds of mutants generated for three CITO generation systems. The total number of mutants generated for NCSystem, SCSystem, and CWRSystem is 290, 287, and 286, respectively. More than 30% of mutants are generated by negate conditionals mutator, which replaces the original conditional with the corresponding negate conditional. For example, '==' is replaced by '! =', '<=' is replaced by '>', and '>=' is replaced by '<'. Mutants generated by void method call mutator account for 28%, which ranks the second.

[1] https://pitest.org/

[2] Methods that can be mutated by Empty Returns Mutator will not be mutated.

Table 3
Information of mutants for CITO generation systems. Column NC, SC and CWR counts the number of different kinds of mutants generated for NCSystem, SCSystem and CWRSystem, respectively.

Type of mutants	Number		
	NC	SC	CWR
FALSE_RETURNS	5	5	5
CONDITIONALS_BOUNDARY	28	27	27
EMPTY_RETURNS	17	17	17
INCREMENTS	16	16	16
VOID_METHOD_CALLS	62	64	63
MATH	31	31	33
NEGATE_CONDITIONALS	106	103	101
NULL_RETURNS	14	13	13
PRIMITIVE_RETURNS	11	11	11
SUM	290	287	286

Table 4
Information of mutants for cycle-breaking module in CITO generation systems. Column NC, SC and CWR counts the number of different kinds of mutants generated for NCSystem, SCSystem and CWRSystem, respectively.

Type of mutants	Number		
	NC	SC	CWR
CONDITIONALS_BOUNDARY	7	6	6
EMPTY_RETURNS	2	2	2
VOID_METHOD_CALLS	4	5	5
MATH	2	2	4
NEGATE_CONDITIONALS	24	21	19
NULL_RETURNS	3	2	2
PRIMITIVE_RETURNS	1	1	1
SUM	43	39	39

Table 4 counts the number of mutants generated for cycle-breaking module, which is the most different module among three CITO generation systems. The number of mutants generated in cycle-breaking module is 43, 39 and 39 for NCSystem, SCSystem, and CWRSystem, respectively. Similar with the whole system, almost half of these mutants are generated by negate conditionals mutator.

Mutation score (MS) is adopted to measure the performance of different MRs. MS indicates the adequacy of a test suite (*TS*) against the program (*P*) under test, which is calculated as follows:

$$MS(P,TS) = \frac{N_k}{N_m - N_e} \tag{3}$$

where N_k counts the number of killed mutants by the test suite, N_m counts the total number of mutants and N_e refers to the number of equivalent mutants, which behave the same as the unmutated program. One kind of output information indicates which module our system executed. Therefore, mutants could be regarded as equivalent mutants if the mutated modules were not executed by the system, or the mutants are generated by mutating the statements that print such indication information. Equivalent mutants are excluded when counting the number of mutants in Tables 3 and 4.

In our experiments, a test suite consists of a set of test cases generated for the same MR. Different from the traditional testing where the expected values are provided, MRs are provided in MT. If the test outputs of the source test case and its follow-up test case violate the given MR for a mutant, the mutant is referred to be killed. The higher the MS, the more adequate the given test suite (i.e., the proposed MR) against the program under the test is.

4.5. Experimental results

We first test three CITO generation systems using the five proposed MRs. All pairs of test cases, i.e., source test cases and follow-up test cases, pass the corresponding CITO generation systems respectively and no MRs are violated. It indicates that no faults can be detected by the five proposed MRs in three CITO generation systems, which increases the likelihood of being valid for our systems. However, the effectiveness of different MRs cannot be distinguished by the three systems. Therefore, we use these systems as benchmarks for the following experiments, that is, to evaluate the effectiveness of the proposed MRs using mutation analysis.

Table 5
Mutation scores of CITO generation systems for different test suites generated based on five MRs. Column N_k counts the number of killed mutants and column MS represents the corresponding mutation scores.

MRs	Type	NCSystem		SCSystem		CWRSystem	
		N_k	*MS*	N_k	*MS*	N_k	*MS*
MR1	None	143	0.4931	141	0.4913	141	0.4930
MR2	ADD	163	0.5621	237	0.8258	238	0.8322
	DEL	146	0.5034	221	0.7700	225	0.7867
	TOTAL	163	0.5621	241	0.8397	242	0.8462
MR3	ADD	221	0.7621	235	0.8188	240	0.8392
	DEL	141	0.4862	210	0.7317	210	0.7343
	TOTAL	229	0.7897	242	0.8432	245	0.8566
MR4	SRC	207	0.7138	205	0.7143	200	0.6993
	TAR	210	0.7241	208	0.7247	203	0.7098
	TOTAL	210	0.7241	208	0.7247	245	0.8566
MR5	None	246	0.8483	247	0.8606	248	0.8671
Total	None	246	0.8483	248	0.8641	248	0.8671

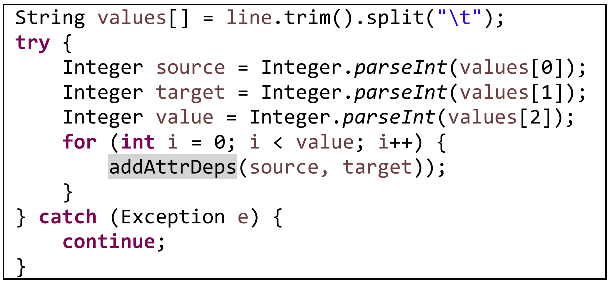
RQ1: What is the performance of different MRs on detecting faulty programs?

For this research question, the generated mutants are used as faulty programs and pairs of test cases are applied in such faulty programs to detect the violation of the proposed MRs. Table 5 shows the MS of CITO generation systems for different test suites generated based on five MRs, where all test cases for the same MR are consisted of a test suite. More than 84% of faulty programs can be detected by all MRs for three CITO generation systems. Specifically, all 177 pairs of test cases killed 246 mutants and detected 84% of faults for NCSystem. All pairs of test cases killed 248 mutants and detected 86% of faults for SCSystem and CWRSystem. Test cases adopted on SCSystem and CWRSystem detected two more faults compared with those applied on NCSystem. These two faults exist in the function for stubbing cost calculation. While in NCSystem, stubbing cost is not calculated when breaking cycles because breaking cycles is based on the number of cycles in which each edge involves. Therefore, these two faults are missed in NCSystem.

Among five MRs, the test suite generated based on MR5 (i.e., combining certain classes with the aim of preserving the class test order for other classes and remaining the same stubbing cost) not only killed the greatest number of mutants, but also killed almost all the mutants that can be killed by other MRs. Although the number of test cases in this test suite is not the greatest (only 10, 8 and 9 test cases for NCSystem, SCSystem, and CWRSystem), the violation detection effectiveness of the test suite for MR5 is the highest.

While for the simplest MR, the permutation of class names (MR1), its test suite included 20 pairs of test cases, but it only killed 143 mutants for NCSystem (141 mutants for SCSystem and CWRSystem). This finding indicates that different MRs own different capability of fault detection, which is consistent with the previous study on MT [21].

More complex MRs are likely to kill more mutants. That is because the follow-up programs generated by more complex MRs, such as MR5, have more different class dependencies from the source programs compared with the follow-up programs generated by simple MRs. For example, Fig. 8 shows a snippet where a faulty program is generated by changing the conditionals boundary "$i < value$" to "$i \leq value$". The function of this snippet is to store the attribute coupling *value* of a class dependence from the *source* class to the *target* class. For instance, if the attribute coupling *value* is five, then the function *addAttrDeps* will be executed five times to add five different attribute coupling. The faulty program generated by conditionals boundary will count one more time



```
String values[] = line.trim().split("\t");
try {
    Integer source = Integer.parseInt(values[0]);
    Integer target = Integer.parseInt(values[1]);
    Integer value = Integer.parseInt(values[2]);
    for (int i = 0; i < value; i++) {
        addAttrDeps(source, target));
    }
} catch (Exception e) {
    continue;
}
```

Fig. 8. A snippet to store attribute coupling values for class dependencies in CITO generation systems.

on attribute coupling for all class dependencies. Different values of attribute coupling require different stubbing cost, and correspondingly affect the generated CITOs. However, such a fault can only be detected by MR5. That is because the follow-up programs generated by MR5 changed the class dependencies to a greater extent compared with other MRs by combining some classes. Specifically, MR1, the permutation of class names, does not affect the class dependencies as well as their attribute coupling. Although MR2 and MR3 add or delete class dependencies for the initial or last class, for the remaining class dependencies, if we sort them by the stubbing cost, the results of sorting are the same as that in the original programs because all class dependencies are added one more attribute coupling. The situation is the same for MR4. The more different class dependencies in follow-up programs, the more difficult for CITO generation systems to preserve the original results. Therefore, a little modification on the original CITO generation systems may generate different results. In other words, the mutants are easy to be killed.

The MS of MR1 is quite low compared with other MRs. As we introduced before, the follow-up programs are constructed by permuting the class names without any modifications on class dependencies, so that in most cases, the CITOs generated for the source program and follow-up program are almost the same except for different notations of classes. Therefore, it is difficult for MR1 to kill more mutants. But MR1 can still detect some faults related to the notation of classes. For example, in NCSystem, if more than one class dependencies are involved in the greatest number of cycles and have the same stubbing cost, the class dependency whose source class owns the smallest index will be removed. A faulty program is generated by mutating the *if* statement, which causes the class dependency with the smallest index of source class is removed, no matter how much the stubbing cost is. Because the indexes of the same class are different in source programs and follow-up programs, the generated CITOs will be different, and such a fault can be detected.

The MS of MR2 on NCSystem is also low compared with that on the other two systems, SCSystem and CWRSystem. Although MR3 and MR2 are similar (both are adding class dependencies with the aim of preserving its source CITO), the MS of MR3 on NCSystem (0.7621) is higher than that of MR2 (0.5621). That is because the changes in the number of cycles for each class dependency by MR2 are quite small. For example, for the source program ATM, MR2 adds three class dependencies 1 → 6, 5 → 6, and 12 → 6 for the initial class 6 while MR3 adds three class dependencies 15 → 3, 15 → 7,15 → 18 for the last class 15. For the follow-up program generated by MR2, all three new class dependencies are not involved in any cycles, and the number of cycles for each class dependency is also unchanged. However, for the follow-up program generated by MR3, the number of cycles for more than 62% (18/29) class dependencies increases. The new class dependency 15 → 18 is also involved in one cycle. In NCSystem, the class dependency involved in the greatest number of cycles is removed. Therefore, the more different class dependencies in follow-up programs, the more difficult for CITO generation systems preserve the source CITOs, and correspondingly, the easier the mutants are killed.

Table 6
Mutation scores of cycle-breaking modules for different test suites generated based on five MRs. Column N_k counts the number of killed mutants and column MS represents the corresponding mutation scores.

MRs	Type	NCSystem		SCSystem		CWRSystem	
		N_k	*MS*	N_k	*MS*	N_k	*MS*
MR1	None	22	0.5116	23	0.5897	23	0.5897
MR2	ADD	28	0.6512	28	0.7179	28	0.7179
	DEL	28	0.6512	28	0.7179	28	0.7179
	TOTAL	28	0.6512	28	0.7179	28	0.7179
MR3	ADD	25	0.5814	28	0.7179	28	0.7179
	DEL	26	0.6047	28	0.7179	28	0.7179
	TOTAL	26	0.6047	28	0.7179	28	0.7179
MR4	SRC	20	0.4651	22	0.5641	22	0.5641
	TAR	20	0.4651	22	0.5641	22	0.5641
	TOTAL	20	0.4651	22	0.5641	22	0.5641
MR5	None	28	0.6512	28	0.7179	28	0.7179
Total	None	28	0.6512	28	0.7179	28	0.7179

Table 7
Mutation scores of CITO generation systems for different types of mutants. Column N_k counts the number of killed mutants and column MS represents the corresponding mutation scores.

Type of mutants	NCSystem		SCSystem		CWRSystem	
	N_k	*MS*	N_k	*MS*	N_k	*MS*
FALSE_RETURNS	4	0.8000	5	1.0000	5	1.0000
CONDITIONALS_BOUNDARY	14	0.5000	14	0.5185	14	0.5185
EMPTY_RETURNS	16	0.9412	15	0.8824	15	0.8824
INCREMENTS	16	1.0000	16	1.0000	16	1.0000
VOID_METHOD_CALLS	56	0.9032	57	0.8906	57	0.9048
MATH	25	0.8065	27	0.8710	30	0.9091
NEGATE_CONDITIONALS	94	0.8868	94	0.9126	91	0.9010
NULL_RETURNS	12	0.8571	12	0.9231	12	0.9231
PRIMITIVE_RETURNS	9	0.8182	8	0.7273	8	0.7273
SUM	246	0.8483	248	0.8641	248	0.8671

For MR2 and MR3, adding and deleting class dependencies show different performance on detecting faulty programs. The test suite generated by adding class dependencies detects more mutants compared with the test suite obtained by deleting class dependencies. This is because the number of test cases generated by deleting class dependencies is much fewer, for example, only three test cases for MR2 and four test cases for MR3 in NCSystem. Although the test suite generated by deleting class dependencies kills fewer mutants, it can still detect faults that are omitted by test cases generated by adding class dependencies. For MR4, the performance of test suites generated by adding a target class or a source class is comparable.

Because cycle-breaking module is the most different part among three systems, Table 6 presents the MS of cycle-breaking modules for different test suites generated based on five MRs. As shown in Table 6, more than 65% of faults in cycle-breaking modules can be detected by all MRs. Similar to the whole system, the test suite generated based on MR5 killed the greatest number of mutants for all three systems. The performance of test suites obtained by MR2 and MR3 are comparable, and it is better than the performance of test suites generated by the relatively weaker MRs, such as MR4 and MR1.

Overall, the proposed MRs are able to detect more than 84% of faults in three CITO generation systems. Among five MRs, MR5 owns the highest capability for fault detection, which killed almost all the mutants that can be killed by other MRs.

RQ2: What is the performance of MT on detecting different kinds of faults?

Table 7 shows MS of three CITO generation systems for different type of mutants. To show the results more intuitively, Figs. 9 to 11

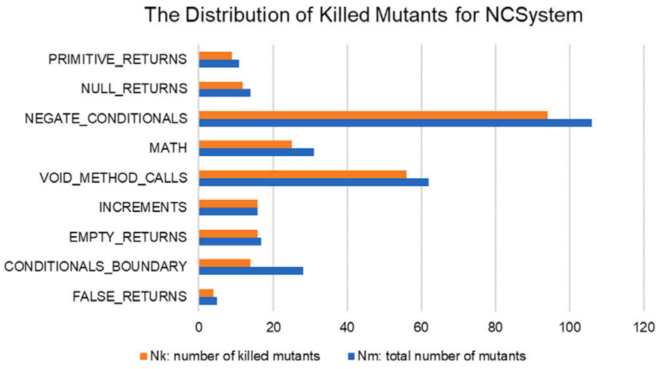


Fig. 9. The distribution of killed mutants for NCSystem.

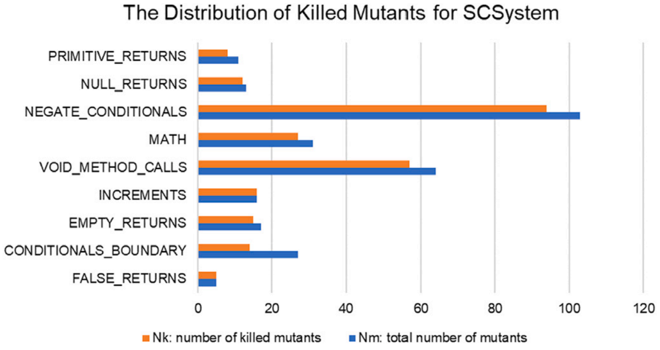


Fig. 10. The distribution of killed mutants for SCSystem.

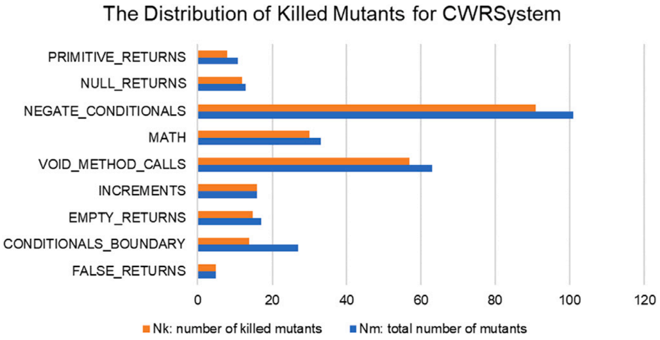


Fig. 11. The distribution of killed mutants for CWRSystem.

present the distribution of these killed mutants for NCSystem, SCSystem, and CWRSystem, respectively.

As shown in Table 7, except for conditionals boundary mutator and primitive returns mutator, more than 80% of mutants generated by other mutators can be killed, especially for increments mutator, all 16 mutants are killed. Over 90% of faults generated by two mutators (empty returns mutator and void method call mutator) can be detected in NCSystem. Similarly, over 90% of faults obtained by two mutators (negate conditionals mutator and null returns mutator) can be detected in SCSystem, and over 90% of faults generated by four mutators (void method call mutator, math mutator, negate conditionals mutator and null returns mutator) can be detected in CWRSystem. Only 50% of mutants generated by conditionals boundary mutator are killed, that is because the relational operator has the same effects on some functions as their boundary counterpart. For example, in function *getMaxStubDep*() which obtains the class dependency with the maximal stubbing cost, the relation operator '>' is used to compare the stubbing cost for two class dependencies and the class dependency with higher stubbing cost will be stored in a temp parameter. In such a case, the effects of '>' and its boundary counterpart '>=' are equal. Similarly, the primitive returns mutator that replaces return values with zero may not violate some proposed MRs. For example, in function *getSCplx*() which calculates stubbing cost for each class dependency, the return value of this function is set to zero by the primitive returns mutator. In such a case, the stubbing cost of any class integration test orders would be zero, which still conforms to some MRs where the stubbing costs of two class test orders generated by the source program and the follow-up program are expected to be equal. The currently proposed MRs are unable to detect such faults. Some other criteria should be provided, such as, the stubbing cost of a generated class integration test order should be greater than zero, which is left for future work.

Overall, more than 80% of mutants generated by seven of nine mutators can be killed. The performance of MT on detecting different kinds of faults is varying, and all mutants can be killed by the increments mutator.

4.6. Discussion

In our experiments, we found that even test cases generated by the same MR can own the different capability to kill mutants. That is because the class dependencies in the generated follow-up programs are different for various test cases. For example, to test NCSystem, test cases #1 and #2 generate two follow-up programs for the source program ATM based on MR3, that is, adding class dependencies for the initial class 15 randomly. The added class dependencies are shown as following:

Follow-up program #1: 15 → 3, 15 → 7, 15 → 12, 15 → 13, 15 → 14, 15 → 18,

Follow-up program #2: 15 → 4, 15 → 6, 15 → 11, 15 → 17, 15 → 19.

Table 8 shows the number of cycles for each class dependency in the source program, and its two follow-up programs #1 and #2, respectively. As shown in Table 8, for those class dependencies that are already involved in cycles in the source program, most of such class dependencies are involved in more cycles in the follow-up program #1. After adding class dependencies, five more class dependencies are involved in cycles in the follow-up program #1 while only two more class dependencies are involved in cycles in the follow-up program #2. It is obvious that class dependencies in the follow-up program #1 are more different from the source program. The number of mutants killed by test case #1 and #2 is 216 and 129, respectively. It is consistent with what we analyzed before, that is, the more different class dependencies in follow-up programs, the more difficult for CITO generation systems to preserve the source CITOs, and correspondingly, the easier the mutants are killed.

We also investigate whether the generated CITOs affect the performance of MT on three CITO generation systems. Table 9 shows the generated CITOs and OCplx by three systems for the four test suites (ANT, ATM, DNS and SPM). Table 10 shows the number of killed mutants by the four test suites respectively. For the program ANT, the numbers of killed mutants are equal for SCSystem and CWRSystem where the generated CITOs are the same. Similarly, for the program SPM, the CITOs generated by NCSystem and CWRSystem are the same, and the numbers of killed mutants are also equal for these two systems. However, for the programs ATM and DNS, although the orders generated by three systems and the corresponding stubbing efforts are equal, the numbers of killed mutants are different, especially for the program ATM. Therefore, we can conclude that the numbers of killed mutants and the orders generated by three programs are not necessarily related.

Table 8
Number of cycles for each class dependency in source program and follow-up programs for ATM.

Class dependencies	Source program	Follow-up program #1	Follow-up program #2
8 → 10	20	33	21
9 → 8	11	18	12
10 → 15	6	25	7
11 → 8, 11 → 9, 11 → 10	5	8	5
12 → 8, 12 → 9, 12 → 10	1	2	1
13 → 8, 13 → 9, 13 → 10			
14 → 8, 14 → 9, 14 → 10			
12 → 11, 13 → 11, 14 → 11	3	6	3
15 → 12, 15 → 13, 15 → 14	/	6	/
15 → 18, 18 → 9	/	1	/
15 → 19, 19 → 9	/	/	1

Table 9
Generated CITOs and OCplx by three systems for four test suites. The number in column CITOs represents the class ID.

Test suites	SUT	CITOs	OCplx
ANT	NCSystem	9 → 12 → 15 → 17 → 3 → 8 → 19 → 21 → 20 → 23 → 2 → 4 → 5 → 16 → 1 → 6 → 7 → 10 → 22 → 25 → 11 → 14 → 24 → 18 → 13	4.269
	SCSystem	4 → 9 → 12 → 15 → 17 → 3 → 5 → 8 → 21 → 20 → 23 → 16 → 1 → 2 → 7 → 10 → 19 → 22 → 6 → 11 → 18 → 24 → 25 → 13 → 14	4.636
	CWRSystem	4 → 9 → 12 → 15 → 17 → 3 → 5 → 8 → 21 → 20 → 23 → 16 → 1 → 2 → 7 → 10 → 19 → 22 → 6 → 11 → 18 → 24 → 25 → 13 → 14	4.193
ATM	/	6 → 16 → 20 → 21 → 1 → 2 → 7 → 17 → 3 → 4 → 5 → 8 → 9 → 11 → 18 → 19 → 10 → 12 → 13 → 14 → 15	2.701
DNS	/	9 → 10 → 13 → 15 → 17 → 27 → 28 → 31 → 36 → 44 → 46 → 49 → 59 → 60 → 61 → 5 → 47 → 53 → 21 → 8 → 14 → 22 → 34 → 39 → 40 → 43 → 58 → 32 → 41 → 1 → 2 → 4 → 6 → 16 → 18 → 19 → 23 → 25 → 26 → 29 → 45 → 48 → 56 → 7 → 11 → 20 → 24 → 30 → 35 → 54 → 55 → 57 → 37 → 42 → 38 → 50 → 51 → 12 → 52 → 3 → 33	1.003
SPM	NCSystem	6 → 11 → 18 → 3 → 10 → 17 → 16 → 2 → 9 → 19 → 1 → 12 → 13 → 14 → 15 → 4 → 5 → 7 → 8	4.361
	SCSystem	6 → 11 → 18 → 4 → 10 → 17 → 3 → 16 → 2 → 9 → 13 → 19 → 1 → 12 → 14 → 15 → 5 → 7 → 8	6.535
	CWRSystem	6 → 11 → 18 → 3 → 10 → 17 → 16 → 2 → 9 → 19 → 1 → 12 → 13 → 14 → 15 → 4 → 5 → 7 → 8	4.111

Table 10
Number of killed mutants by different test suites.

Test suites	SUT	N_k
ANT	NCSystem	234
	SCSystem	242
	CWRSystem	242
ATM	NCSystem	234
	SCSystem	241
	CWRSystem	240
DNS	NCSystem	237
	SCSystem	233
	CWRSystem	233
SPM	NCSystem	245
	SCSystem	241
	CWRSystem	245

The more complicated MRs, the stronger the detection capability is. It indicates that more complex MRs that can validate the CITO generation systems may exist. Therefore, except for the five MRs introduced explicitly before, we also present an example for a potentially immature MR, division, which is inspired from MR5 (combining classes with the aim of preserving the original test order). The potentially new MR is dividing some existing classes with the aim of preserving the original test order. Fig. 12 shows an example of this new MR on NCSystem. The NCSystem breaks cycles by removing the class dependency that is involved in the greatest number of cycles. Let *SP* denote the source program that contains *six* classes, and the class integration test order generated for the program *SP* is O_{sp}, which is further supposed to be $[C_4, C_5, C_3, C_2, C_1, C_6]$. A follow-up program *FP* can be generated by dividing the existing class C_4 into two classes (C_4 and C_4') and remaining the existing class dependencies $C_3 \rightarrow C_4$, $C_4 \rightarrow C_1$ and $C_5 \rightarrow C_4$. In the follow-up program, the division of class C_4 does not affect the CITO generated by approaches, therefore, the class integration test order generated for the program *FP* is similar with O_{sp}, that is, $O_{fp} = [C_4, C_4', C_5, C_3, C_2, C_1, C_6]$. We generated test suites based on this sample program to validate the NCSystem. Although no MRs are violated, test cases generated by six MRs killed 251 mutants for NCSystem and achieved 0.8655 in mutation score. The test cases generated by the new MR improve the MS by 2%. More complex MRs that can validate the CITO generation systems may exist, which needs our more exploration in the future.

Although the proposed MRs focus on CITO generation systems, these MRs can also be extended to other cycle-breaking algorithms on directed weighted graphs. The proposed MRs can be summarized as four types, that is, permutation (MR1), edge adding/deleting (MR2 and MR3), node adding (MR4) and combination (MR5), which can be used to generate specific MRs for other cycle-breaking problem. For example, the feedback vertex set (FVS) problem is to search for a set of vertices whose removal leaves a graph without cycles and the sum of weights for such vertices is minimal. The FVS problem is similar to the CITO generation problem except that FVS problem is to remove vertices. However, four types of MRs can also be generated to test the algorithms for FVS problem. For a given source graph G and its solution S, the follow-up graph G_F can be generated by permuting the notation of nodes, adding an edge to the node in solution S, adding a node that links to all other remaining nodes with a very large weight, and so on. The solution S_F for the follow-up graph G_F should be the same as the original solution S.

4.7. Threats to validity

Although the experimental results demonstrate the effectiveness of the proposed MRs, it still faces the following threats to validity.

For the internal validity, we implemented three CITO generation systems for the experiments because no such systems are public. To

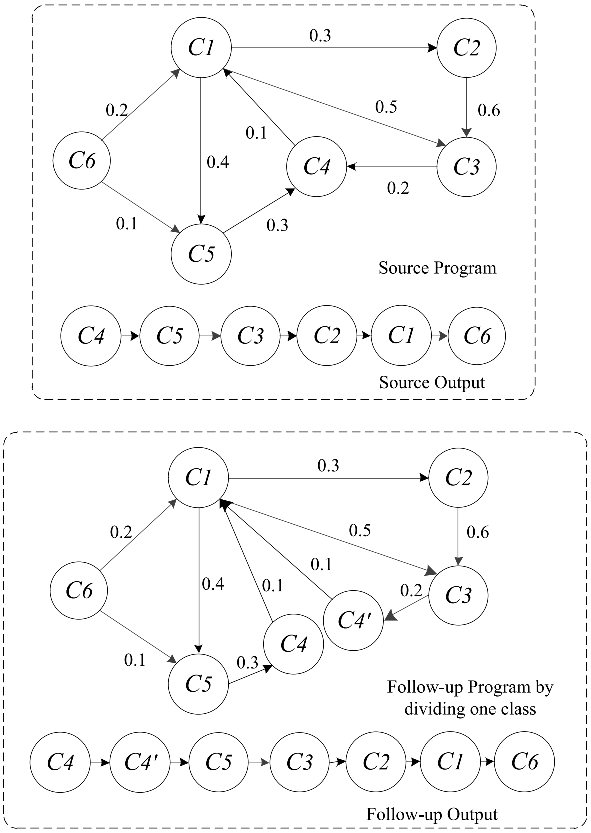


Fig. 12. Example for new MR (division) performed on NCSystem. Class C_4 is divided into two classes (C_4 and C_4').

check the correctness of our implementation, we generate test suites based on the proposed MRs. Our implemented systems pass all test suites successfully and no MRs are violated.

For the external validity, more than 84% of mutants were killed by around 117 pairs of test cases for three CITO generation systems. As we analyzed in Discussion, even test cases generated by the same MR can own the different capability to kill mutants. To improve the fault detection capability, test cases should be able to generate more different class dependencies in follow-up programs. We intend to explore how to design such test cases in the future.

In addition, to the best of our knowledge, it is the first time to validate CITO generation systems, which means no other testing or validation methods for CITO generation systems exist. Therefore, it is impossible to compare the effectiveness of MT with other kinds of testing or validation methods for CITO generation systems.

5. Conclusion

Previous studies focus on exploring different kinds of approaches for CITO problem but omit to validate such CITO generation systems. A test oracle problem exists in these CITO generation systems because there are no practically feasible ways to validate their generated class integration test orders, which are normally not global optima. To address this issue, we adopt the technique of MT to validate CITO generation systems. In this paper, we propose five kinds of MRs for the validating of these systems. Although no faults can be detected by the test cases generated based on the five proposed MRs, the likelihood of being valid increases for the CITO generation systems. We also conducted mutation analysis to evaluate the effectiveness of five proposed MRs. The experimental results show that more than 84% of faulty programs can be detected by all MRs for three CITO generation systems. The fault detection capability of the proposed MRs is different. The more complicated MRs, the stronger the detection capability is. Therefore, more complex MRs that can validate the CITO generation systems may exist, which requires our more exploration in the future.

Acknowledgments

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong, China (No. 11208017) and the research funds of City University of Hong Kong (7005028 and 7005217), and the Research Support Fund by Intel, China (9220097), and funding supports from other industry partners, China (9678149, 9440227, 9229029, 9440180 and 9220103).

References

[1] H. Melton, E. Tempero, An empirical study of cycles among classes in Java, Empir. Softw. Eng. 12 (4) (2007) 389–415.

[2] D. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, A test strategy for object-oriented programs, in: Proceedings of the 19th Annual International Computer Software and Applications Conference, COMPSAC'95, 1995, pp. 239–244.

[3] L.C. Briand, J. Feng, Y. Labiche, Experimenting with Genetic Algorithms to Devise Optimal Integration Test Orders, Tech. Rep. TR SCE-02-03, Carleton University, 2003, pp. 204–234.

[4] T.Y. Chen, S.C. Cheung, S.M. Yiu, Metamorphic Testing: A New Approach for Generating Next Test Cases, Tech. Rep. HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology, 1998.

[5] L.C. Briand, Y. Labiche, Y. Wang, An investigation of graph-based class integration test order strategies, IEEE Trans. Softw. Eng. 29 (7) (2003) 594–607.

[6] N.L. Hashim, H.W. Schmidt, S. Ramakrishnan, Test order for class-based integration testing of Java applications, in: Proceedings of the 5th International Conference on Quality Software, QSIC'05, 2005, pp. 11–18.

[7] P. Bansal, S. Sabharwal, P. Sidhu, An investigation of strategies for finding test order during Integration testing of Object-Oriented applications, in: Proceedings of International Conference on Methods and Models in Computer Science, ICM2CS, 2009, pp. 1–8.

[8] A. Abdurazik, J. Offutt, Using coupling-based weights for the class integration and test order problem, Comput. J. 52 (5) (2009) 557–570.

[9] F.T. Chan, T.Y. Chen, S.C. Cheung, M.F. Lau, S.M. Yiu, Application of metamorphic testing in numerical analysis, in: Proceedings of the IASTED International Conference on Software Engineering, 1998, pp. 191–197.

[10] T.Y. Chen, F. -C. Kuo, H. Liu, P. -L. Poon, D. Towey, T.H. Tse, Z.Q. Zhou, Metamorphic testing: A review of challenges and opportunities, ACM Comput. Surv. 51 (1) (2018) 1–27.

[11] S. Segura, G. Fraser, A.B. Sanchez, A. Ruiz-Cortés, A survey on metamorphic testing, IEEE Trans. Softw. Eng. 42 (9) (2016) 805–824.

[12] L.L. Pullum, O. Ozmen, Early results from metamorphic testing of epidemiological models, in: Proceedings of the 2012 ASE/IEEE International Conference on BioMedical Computing, BioMedCom'12, 2012, pp. 62–67.

[13] A. Ramanathan, C.A. Steed, L.L. Pullum, Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics, in: Proceedings of the 2012 ASE/IEEE International Conference on BioMedical Computing, BioMedCom'12, 2012, pp. 68–73.

[14] M. Lindvall, D. Ganesan, R. Árdal, R.E. Wiegand, Metamorphic model-based testing applied on NASA DAT: An experience report, in: Proceedings of the 37th International Conference on Software Engineering, ICSE'15, 2015, pp. 129–138.

[15] J. Ding, X. Hu, V. Gudivada, A machine learning based framework for verification and validation of massive scale image data, IEEE Trans. Big Data (2017) 1.

[16] T.Y. Chen, F. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, Z.Q. Zhou, Metamorphic testing for cybersecurity, Computer 49 (6) (2016) 48–55.

[17] S. Pugh, M.S. Raunak, D.R. Kuhn, R. Kacker, Systematic testing of post-quantum cryptographic implementations using metamorphic testing, in: Proceedings of the 4th International Workshop on Metamorphic Testing, MET'19, 2019, pp. 2–8.

[18] N. Mouha, M.S. Raunak, D.R. Kuhn, R. Kacker, Finding bugs in cryptographic Hash function implementations, IEEE Trans. Reliab. 67 (3) (2018) 870–884.

[19] V. Le, M. Afshari, Z. Su, Compiler validation via equivalence modulo inputs, in: Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI'14, 2014, pp. 216–226.

[20] Q. Tao, W. Wu, C. Zhao, W. Shen, An automatic testing approach for compiler based on metamorphic testing technique, in: Proceedings of the 2010 Asia Pacific Software Engineering Conference, APSEC'10, 2010, pp. 270–279.

[21] H. Liu, F. Kuo, D. Towey, T.Y. Chen, How effectively does metamorphic testing alleviate the oracle problem? IEEE Trans. Softw. Eng. 40 (1) (2014) 4–22.