Information and Software Technology 129 (2021) 106438

Contents lists available at ScienceDirect

Information and Software Technology

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

Evaluating the effects of similar-class combination on class integration test order generation

Miao Zhang [a], Jacky Wai Keung [a], Yan Xiao [b,*], Md Alamgir Kabir [a]

[a] Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong, China
[b] School of Computing, National University of Singapore, 117417, Singapore

ARTICLE INFO

Keywords:
Class integration test order
Integration testing
Similar classes
Stubbing complexity

ABSTRACT

Context: In integration testing, the order in which classes are integrated and tested is significant for the construction of test stubs. With the existing approaches, it is usually difficult to generate the sub-optimal test orders for real applications, which have large numbers of classes.

Objective: There exist moderately large numbers of classes in software systems, which is one of the main factors that complicate the generation of class integration test order (CITO). The main objectives of this study are reducing the problem space for CITO generation, and minimizing the stubbing cost of the generated test orders.

Method: The approach proposed in this study is based on the hypothesis that similar-class combination can remove class dependencies and yield a smaller problem space. Identical class dependence and symmetric classes are the two main properties that are used to identify similar classes. In addition, a new cycle-breaking algorithm is introduced to minimize the stubbing cost of the generated test orders, which fully considers the two factors (number of test stubs and the corresponding stubbing complexity) that affect the overall stubbing cost. Empirical experiments are conducted on nine open-source Java programs to evaluate the performance of the proposed approach.

Results: With similar-class combination, the proposed approach reduced the numbers of classes and class dependencies by over 10% and 6%, respectively, for six programs. Moreover, for four programs, the proposed approach reduced the number of cycles among class dependencies by more than 20%. The cycle-breaking algorithm achieved reduction of more than 13% in the stubbing cost, thus outperforming other competing techniques.

Conclusions: The proposed method relies on the two aforementioned important properties to identify similar classes, and these properties are known to significantly improve the performance of CITO generation. The results obtained in this study confirmed the capability of the proposed approach in terms of minimizing the number of classes and class dependencies in programs. It outperformed other competing methods in minimizing the stubbing costs of the generated test orders.

1. Introduction

In the integration testing of a program, it is important to determine the order in which classes are integrated and tested. Class integration test order (CITO) affects the preparation of test cases, and it determines the order in which inter-class faults are detected [1]. When the classes in a program are independent of each other, the testing order is arbitrary, and the classes are tested directly and separately. However, in real applications, class dependency is commonly encountered, for example, when class *A* depends on class *B*, we integrate class *B* before class *A*. If we integrate and test class *A* first, class *B* may be unreliable or be unable to provide related services to class *A*. In such a case, a test stub must be created for class *A* to emulate the services of class *B*. Especially in large-scale programs, stubbing is inevitable because circular dependencies are quite common [2].

Although many tools to easily create stubs (or mocks) have been implemented, for instance, Mockito [3] and JUnit [4], test stubs are error-prone [5]. Therefore, designing an optimal test order with the minimal stubbing cost is extremely important.

Searching for the globally optimal class test order is impractical, even when the number of classes is moderately large. Graph-based and

* Corresponding author.
E-mail addresses: miazhang9-c@my.cityu.edu.hk (M. Zhang), jacky.Keung@cityu.edu.hk (J.W. Keung), dcsxan@nus.edu.sg (Y. Xiao), makabir4-c@my.cityu.edu.hk (M.A. Kabir).

https://doi.org/10.1016/j.infsof.2020.106438
Received 27 December 2019; Received in revised form 29 July 2020; Accepted 22 September 2020
Available online 25 September 2020

search-based approaches have been extensively adopted to search for the locally optimal class test order instead [6]. Graph-based approaches aim to break cycles among class dependencies, while search-based approaches devise superior class test orders by executing evolutionary operations, such as mutation and crossover in a genetic algorithm, on the initial test order. However, the performances of these two types of approaches are limited by two problems:

- It becomes difficult for the existing heuristics to generate the locally optimal test order for real applications with a moderately large number of classes.
- Ignorance of the factors that affect the overall stubbing cost makes these methods ineffective.

The first problem affects the two types of approaches in different ways. For instance, the performance of graph-based methods deteriorates when applied to a program with an enormous number of cycles. For example, the program Byte Code Engineering Library [7] from the Jakarta project is used to analyze, create, and manipulate binary Java class files, and it contains more than 400,000 cycles. With the existing graph-based approaches, it is difficult to manage such a large number of cycles. The unsatisfactory performance of search-based methods is caused by the underlying encoding strategy for class test orders. Given an integer corresponding to a class, a vector of such integers represents a class test order. For a program containing N classes, the search space contains $N!$ possible solutions, which is an enormous number in the case of a program with hundreds or thousands of classes.

We aim to reduce the problem space to address the first issue. The CITO generation is basically equivalent to the problem of the identification of a minimal feedback vertex set (FVS) [8]. To simplify the problem of finding an FVS, Orenstein et al. [9] proposed a new kind of graph reduction operation in which they eliminate edges from the graph, such that an optimum solution for the reduced graph yields an optimum solution for the original graph. To this end, we observe that a few classes are involved in similar cycles. If we regard such "similar classes" holistically, some cycles containing such classes are redundant and can be removed, which reduces the problem space. Therefore, we hypothesize that *similar-class combination can reduce the problem space for large-scale programs that contain a moderately large number of classes*. Based on this hypothesis, two properties pertaining to class dependency, namely, identical class dependence (ICD) and symmetric classes (SC), are proposed to identify similar classes. Owing to the combination of similar classes, overall, the method is required to manage fewer classes, which reduces the search space for the search-based approaches. Similarly, for the graph-based approaches, the numbers of class dependencies and cycles are reduced.

In terms of the second problem, as suggested by Briand et al. in [7], "more precise indicator" rather than the number of test stubs should be used to assess the stubbing cost, therefore, they proposed the overall stubbing cost to measure the effort required to construct all of the test stubs during integration testing [10]. The overall stubbing cost is affected by two factors: the distinct number of test stubs and the corresponding stubbing complexity (which is a measure of the stubbing cost of each test stub). Most existing graph-based approaches are unable to determine a satisfactory class test order because they do not simultaneously consider these two factors. For example, Briand et al. [7] assigned weights to estimate the number of cycles that each class dependency is involved in and constructed test stubs to emulate the class dependency with the maximal value of a weight; in this manner, they only considered the first factor that impacts the stubbing cost. Hashim et al. [11] focused only on the second factor and sequentially created test stubs for the class dependencies with the minimal stubbing costs. The same problem is encountered in the search-based approaches due to the lack of sufficient guidance in the search process.

Therefore, we propose a cycle-breaking algorithm to simultaneously consider these two factors that affect the overall stubbing cost. Given a set of class dependencies that are going to be removed, this algorithm finds superior alternatives that break the same cycles or an equal number of cycles with lower stubbing cost than the original class dependencies.

We evaluate the proposed approach by conducting experiments on nine open-source Java programs. The results indicate that the proposed approach reduces the number of cycles and the number of classes for eight of the nine programs. In the case of six of the nine programs, the numbers of classes and class dependencies are reduced by more than 10% and 6%, respectively. Moreover, for four programs, the proposed approach reduces the number of cycles by more than 20%. In generating the class test order with the minimum stubbing cost, the proposed approach reduces the stubbing cost for six programs by 0.35% to 13.09%, and its performance is at least comparable to if not better than those of the existing approaches. Overall, similar-class combination does not degrade the performance of the cycle-breaking techniques when considering the coupling values of class dependencies. The main contributions of this paper are as follows:

- We analyze class dependencies and propose two properties, namely ICD and SC, to identify similar classes. Two propositions are elucidated to verify the effectiveness of similar-class combination.
- We apply the idea of similar-class combination to CITO generation to minimize the search space for search-based methods and the number of cycles for graph-based methods.
- We develop a graph-based approach for class test order generation, which includes a cycle-breaking algorithm to comprehensively consider the two factors, namely number of test stubs and the corresponding stubbing complexity, that affect the overall stubbing cost.
- A set of experiments is conducted to validate the effectiveness of the proposed approach and the effects of similar-class combination on the CITO generation performance.

The remainder of this paper is organized as follows. Section 2 introduces the background, motivation and problem representation of the proposed approach. We present our approach in Section 3. The experiments follow in Section 4. Related work is presented in Sections 5 and 6 concludes this work.

2. Background and motivation

This section first presents preliminary knowledge about CITO generation, including test stubs and stubbing complexity. Then, a sample program is used to illustrate our motivation. Finally, problem representation is provided.

2.1. Background

A test stub describes the coupling between the source class and the target class on which the source class depends, providing services from the target class to the source class. Attribute assessment and method invocation are two common routes for class interaction, and they are termed attribute coupling and method coupling, respectively.

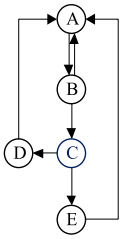
Fig. 1 shows a source code example to describe the coupling information between class A and class B. In Fig. 1, the attribute coupling between class A and class B is one, because class A accesses an instance of class B as the parameter for $methodA2$ (line 10). The method coupling between class A and class B is three, i.e., the constructor of class A invokes the constructor of class B (line 3), $methodA1$ invokes method $getInt()$ (line 8), and $methodA2$ invokes method $getDouble()$ in class B (line 11).

Briand et al. [10] proposed the concept of stubbing complexity to measure the stubbing cost of a test stub (i, j). This idea depends on the attribute coupling and method coupling, which count the number of

```
1  public class A {
2    public A() {
3      B b = new B();
4      int a1 = 1;
5      double a2 = 0.1;
6    }
7    public int methodA1() {
8      return b.getInt();
9    }
10   public double methodA2(B b) {
11     a2 = a1 + b.getDouble();
12     return a2;
13   }
14 }
15
```

```
1  public class B {
2    public B() {
3      int b1 = 1;
4      double b2 = 1.0;
5    }
6    public int getInt(){
7      return b1;
8    }
9    public double getDouble() {
10     return b2;
11   }
12 }
```

Fig. 1. A source code example to describe the coupling information between class A and B.



(a) ORD of Sample

Class Deps	Attribute Coupling & Method Coupling	# of Cycles	Stubbing Complexity	Cycles-Weight Ratio
A→B	(5, 5)	3	1	3
B→A	(0, 2)	1	0.283	3.534
B→C	(1, 2)	2	0.316	6.329
C→D	(0, 1)	1	0.141	7.092
D→A	(3, 3)	1	0.6	1.667
C→E	(1, 3)	1	0.447	2.237
E→A	(2, 0)	1	0.283	3.534

(b) Information of Class Dependencies in Sample

Fig. 2. A sample program to illustrate our motivation.

attribute accesses and method invocations, respectively. The stubbing cost is calculated using Eq. (1).

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

Because attribute coupling and method coupling may vary significantly within the same program, we normalize them using min–max normalization [12] to avoid the influences of the extreme values. In Eq. (1), the normalized attribute coupling and method coupling are linear-weighted, and their weights are equal to 0.5.

For a generated CITO o, the overall stubbing complexity (OCplx) is used to estimate the overall stubbing cost through integration testing. OCplx is equal to the sum of the stubbing complexity of each test stub that belongs to the set $Stubs$, and it is calculated as follows.

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

According to Eq. (2), the overall stubbing cost is affected by two factors: (1) The number of test stubs or the number of class dependencies removed. As the number of class dependencies removed increases, the stubbing cost possibly increases. (2) The corresponding stubbing complexity, that is, the overall stubbing cost tends to decrease if the test stubs for all of the removed class dependencies have relatively low stubbing costs.

2.2. Motivation

In this subsection, we will use a sample program to illustrate why the existing approaches are unable to obtain an optimal class test order. Then, we present our approach to address this problem.

Fig. 2(a) shows the object relation diagram (ORD) of a sample program consisting of five classes A, B, C, D and E, which each node represents a class, and a dependency from the source class to the target class is represented by a directed edge from the head node to the tail node. According to Fig. 2(a), this sample program contains three cycles.

(1) $A \rightarrow B \rightarrow A$
(2) $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$
(3) $A \rightarrow B \rightarrow C \rightarrow E \rightarrow A$

Table 1
Comparison of results for cycle-breaking strategies.

Strategies	Removed Deps	Stubs	OCplx
NC	$A \rightarrow B$	1	1
SCplx	$C \rightarrow D$, $B \rightarrow A$, $E \rightarrow A$	3	0.707
CWR	$C \rightarrow D$, $B \rightarrow A$, $E \rightarrow A$	3	0.707
Optimal	$B \rightarrow C$, $B \rightarrow A$	2	0.599

Table 2(b) summarizes the information pertaining to class dependencies in this sample program, including the value of couplings, number of cycles involved, stubbing complexity calculated using Eq. (1), and the cycles–weight ratio. For example, class dependency $A \rightarrow B$, both its attribute coupling and method coupling are five, and its cycles–weight ratio is three (3/1 = 3).

To generate the class test orders for this sample program, we first eliminate all of the cycles among the class dependencies. The existing graph-based approaches can be characterized into three types based on their cycle-breaking strategies.

- Number of Cycles (NC): Deleting the class dependency involved in the greatest number of cycles. For instance, Briand et al. [7] used the product of in-degree of the source node and out-degree of the target node as edge weight in the ORD to emulate the number of cycles in which the class dependency is involved.
- Stubbing Complexity (SCplx): Deleting the class dependency that requires the minimal stubbing effort. For example, Hashim et al. [11] assigned the value of coupling measures to each class and measured the effort required to construct the test stubs for such classes.
- Cycles–Weight Ratio (CWR): Deleting the class dependency with the highest cycles–weight ratio value, such as the approaches proposed by Bansal et al. [13] and Abdurazik et al. [14]. The class dependency with the highest cycles–weight ratio value indicates that it is involved in the greatest number of cycles, but incurs a minimal stubbing cost if stubbed.

Table 1 presents the class dependencies removed by the above three typical cycle-breaking strategies, as well as the number of test stubs and the overall stubbing complexity calculated using Eq. (2). The optimal choices for class dependency removal are also listed.

As shown in the results, none of the overall stubbing costs of NC, SCplx or CWR are the minimal values. NC only considers the number of removed dependencies, while SCplx only compares stubbing complexity. Even in the case of CWR, although it considers these two factors that affect the overall stubbing cost, it can still lead to sub-optimal results owing to rough calculation. For example, the dependency $B \rightarrow C$ is involved in two cycles with the stubbing complexity (0.316), while the dependency $C \rightarrow D$ is involved in only one cycle with the stubbing complexity (0.141). Accordingly, constructing a test stub for the dependency $C \rightarrow D$ seems to be a good choice because its CWR (7.092) is slightly higher than that of $B \rightarrow C$ (6.329), but the results indicate that removing the dependency $B \rightarrow C$ rather than $C \rightarrow D$ is the best choice.

The optimal solution is to remove dependencies $B \rightarrow C$ and $B \rightarrow A$ to break cycles, even though these two dependencies do not have the best values in terms of the number of cycles, stubbing complexity or cycles–weight ratio.

Similar scenarios are encountered in other types of approaches based on evolutionary algorithms. For example, genetic algorithms [10, 15] regarded all test orders as the population and the overall stubbing complexity as the fitness. However, these algorithms do not consider any detailed factors (such as, the stubbing complexity of each class dependency) that affect the overall stubbing cost as the guidance information to generate new test orders. Several multi-objective evolutionary algorithms [6,16] consider only the number of emulated methods and attributes, but they omit the detailed information. For

Table 2
Information of used programs.

Programs	Source		Description	Classes
daisy	https://sir.csc.ncsu.edu/php/common/download.php?ac=pub&key=sir/java objects&id=51&file=daisy_1.1.tar.gz	(v1.1)	NFS UNIX-like file system	23
deos	https://sir.csc.ncsu.edu/php/common/download.php?ac=pub&key=sir/java objects&id=57&file=deos_1.1.tar.gz	(v1.1)	A scheduler from a real-time executive for avionics systems	25
email_spl	https://sir.csc.ncsu.edu/php/common/download.php?ac=pub&key=sir/java objects&id=157&file=email_spl_1.0.tar.gz	(v1.0)	Email tool	39
GPL_spl	https://sir.csc.ncsu.edu/php/common/download.php?ac=pub&key=sir/java objects&id=158&file=GPL_spl_1.0.tar.gz	(v1.0)	Graph generator	178
JHotDraw	http://sourceforge.net/projects/jhotdraw/ package used: org.jhotdraw.draw of JHotDraw (v7.5.1)		2D graphics framework	411
jmeter	https://sir.csc.ncsu.edu/php/common/download.php?ac=pub&key=sir/java objects&id=8&file=jmeter_1.0.tar.gz	(v1.0)	Load test tool	372
log4j3	https://sir.csc.ncsu.edu/php/common/download.php?ac=pub&key=sir/java objects&id=106&file=log4j3_1.2.tar.gz	(v1.2)	Log tool for Java	261
MyBatis	http://code.google.com/p/mybatis/ MyBatis (v3.0.2)		Java persistence framework	428
notepad_spl	https://sir.csc.ncsu.edu/php/common/download.php?ac=pub&key=sir/java objects&id=162&file=notepad_spl_1.0.tar.gz	(v1.0)	Source code editor	65

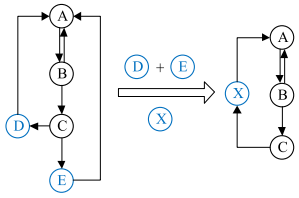
Table 3
Number of instances for similar classes identified by two properties (ICD and SC).

Programs	# of instances identified by ICD	# of instances identified by SC
daisy	1	1
deos	0	0
email_spl	5	3
GPL_spl	16	11
JHotDraw	26	18
jmeter	15	5
log4j3	16	15
MyBatis	21	16
notepad_spl	5	3
Total	105	72

instance, to minimize the number of emulated methods, the number of test stubs can be reduced, or the test stubs for those class dependencies with fewer method invocations can be created.

Therefore, in this paper, we conduct an analysis involving the number of test stubs and their corresponding stubbing complexity when breaking cycles. For this example, we have three choices: (1) to remove a class dependency involved in all three cycles to minimize the test stub number, i.e., $A \rightarrow B$, (2) to remove the class dependency with the lowest stubbing cost in each cycle, i.e., $B \rightarrow A$, $C \rightarrow D$ and $E \rightarrow A$, and (3) to remove the class dependency involved in only a part of the cycles, and remove the class dependency with the lowest stubbing cost in each cycle for the remaining cycles, i.e., $B \rightarrow C$ and $B \rightarrow A$. Then, we compare the three choices and select the best solution with the lowest stubbing cost. These three choices have the same effect, that is, breaking the same cycles, and we search for the choice that has the lowest stubbing cost. Therefore, in the proposed algorithm, we measure the stubbing cost of the current test order, and decide whether to remove a class dependency involved in more cycles that minimizes the test stub number, or to create a test stub for a class dependency with the minimal stubbing cost that reduces the final stubbing cost.

Meanwhile, we aim to combine similar classes to reduce the problem space for the existing approaches. Fig. 3 shows an example of similar-class combination for the sample program in Fig. 2. As shown in Fig. 2(a), classes D and E depend only on class A, and class C depends on classes D and E. Classes D and E are "similar" because they are involved in the same cycles. Therefore, we combine these two classes and replace them with a new class X. The ORD for the "new" sample program is shown on the right side of Fig. 3(a). The class number and cycle number of the "new" sample program decrease by one. The information of the "new" sample program is shown in Fig. 3(b). Different from the original sample program, SCplx and CWR can also generate the optimal solution for the reduced program.



(a) Example for Similar Class Combination

Class Deps	# of Cycles	Stubbing Complexity	Cycles-Weight Ratio
A→B	2	1	2
B→A	1	0.283	3.534
B→C	1	0.316	3.165
C→X	1	0.588	1.701
X→A	1	0.883	1.133

(b) Information for the "New" Sample Program

Fig. 3. An example for similar class combination. Class D and E are similar because they are involved in the same cycles. A new class X is created to replace them.

2.3. Problem representation

Class Integration Test Order (CITO): o is a class integration test order generated for a given program containing m classes, i.e., $o = \{C_1, C_2, \ldots, C_m\}$.

Let O represents all potential solutions for inter-class integration testing, $O = \{o_1, o_2, \ldots, o_n\}$, where $n = m!$. Each solution o_j is a test order from O, $o_j = \{C_{j1}, C_{j2}, \ldots, C_{jm}\}$ where C_{ji} refers to a class that will be integrated and tested in the ith position in o_j.

Class Integration Test Order (CITO) Generation Problem: CITO generation problem aims to find a solution $o_s = \{C_{s1}, C_{s2}, \ldots, C_{sm}\}$ such that: $Ocplx(o_s) \leq Ocplx(o_j)$, where $Ocplx\ (o_j)$ is an objective function that represents the overall stubbing complexity of the class integration test order o_j.

For graph-based methods, the critical step in CITO generation problem is cycle-breaking. Let c represents the number of all of the cycles among class dependencies in a given program. A set of class dependencies Dep, are removed to break c cycles, $Dep = \{dep_1, dep_2, \ldots, dep_k\}$ where $k \leq c$. Correspondingly, a set of test stubs are constructed for Dep, and a class integration test order o is generated. Specifically, for this kind of approaches, CITO generation problem aims to search for a set of class dependencies $Dep_s = \{dep_{s1}, dep_{s2}, \ldots, dep_{sk}\}$ such that: the generated class integration test order o_s can satisfy $Ocplx(o_s) \leq Ocplx(o_j)$ where o_j represents any other class integration test orders.

3. Approach

This section introduces the proposed CITO generation approach. Fig. 4 shows an overview of the proposed approach, which takes a program containing ten classes as an example to make the process of CITO generation more intuitively. We first construct an ORD for the given program, then combine similar classes to reduce the graph.

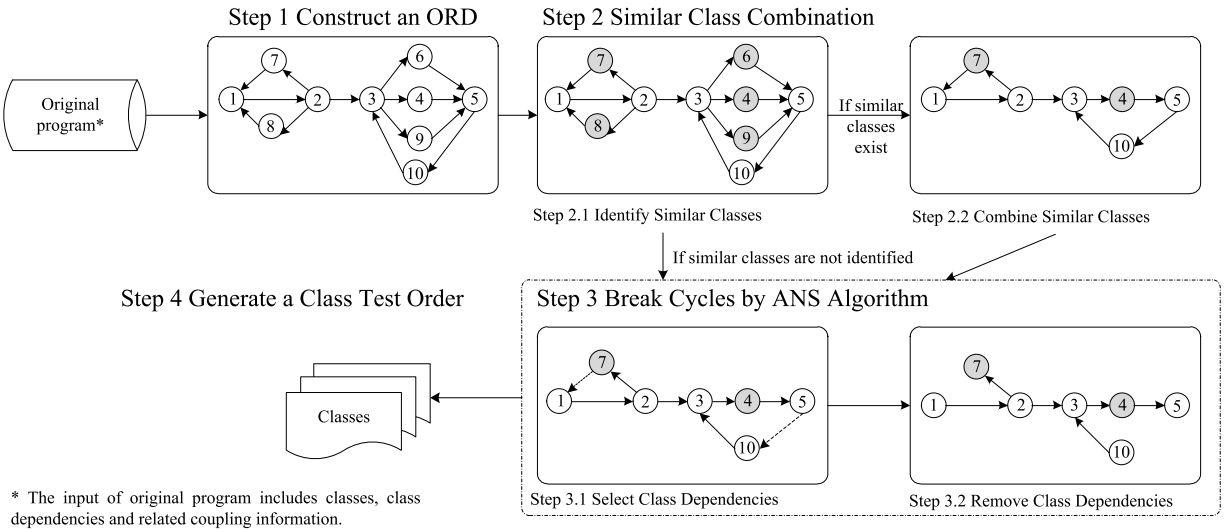


Fig. 4. Overview of our approach.

Our hypothesis is that *similar-class combination can reduce the problem space of large-scale programs that contain a moderately large number of classes*. Therefore, a new notion of similar classes has been developed through the identification of two properties pertaining to class dependence, namely, ICD and SC. Through similar-class combination, some redundant cycles containing similar classes can be effectively identified and removed. Thereafter, a "reduced" graph that contains fewer class dependencies and class cycles is generated.

Next, Tarjan's algorithm [17] is adopted to divide the diagram into different strongly connected components. We identify all of the cycles in each strongly connected component.

The new proposed cycle-breaking algorithm **ANS**, which aims to keep a good **A**nalysis involving the **N**umber of removed dependencies and the related **S**tubbing complexity, is applied to each strongly connected component. In our algorithm, we first select a class dependency as an initialization. Then, we search for a superior choice (i.e., deleting other dependencies) that has a similar effect as removing the initial dependency, such as breaking the same cycles, but incurs a lower stubbing cost. The above process is repeated until no cycles remain in the strongly connected component.

Finally, test stubs are constructed for the removed dependencies, and a CITO is generated for the acyclic diagram.

Similar-class combination and cycle-breaking algorithm are the main components of our approach, which differs from other methods. The details of these components are discussed in the following subsections

3.1. Similar-class combination

In this section, we analyze the relationships among classes and propose two properties, namely identical class dependence (ICD) and symmetric classes (SC), to identify similar classes. Moreover, two propositions are presented to verify that similar-class combination can reduce the problem space of CITO generation. Proposition 1 demonstrates that classes having the ICDs are involved in the same cycles, and Proposition 2 shows that SCs have the same test priority. Classes with the same test order or classes that are involved in the same cycles can be combined to reduce the numbers of classes and class dependencies. We first describe the following notations that are used to describe the proposed properties for a given class C.

- Set *Target_C* stores all classes upon which class C depends.
- Set *Source_C* is the set of dependent classes of class C, which stores all classes that depend on class C.
- Set *Cycles_C* stores a set of cycles among class dependencies that class C is involved in.
- List *Order_C* denotes the final test order of class C.

In this paper, we define the concepts ICD and SC and subsequently propose two corresponding propositions to demonstrate the effectiveness of these concepts:

Definition 1 (*Identical Class Dependence*)**.** Given two classes C_i and C_j, if they depend on the same set of classes, and their dependent classes are equal, i.e., $Target\_C_i = Target\_C_j \bigwedge Source\_C_i = Source\_C_j$, then classes C_i and C_j are said to be having identical class dependence.

Proposition 1. *For two classes C_i and C_j having identical class dependence, if class C_i is involved in cycles among class dependencies, then C_j is also involved in the same cycles, which means that all classes except for class C_i or C_j in these cycles are equal, i.e., $Cycles\_C_i = Cycles\_C_j$.*

Proof. Suppose the given conclusion is false; that is, the set of cycles containing class C_i is not the same as the set of cycles containing class C_j, i.e., $Cycles\_C_i \neq Cycles\_C_j$, which includes three situations:

(1) The set $Cycles\_C_i$ is a subset of the set $Cycles\_C_j$, or vice versa, i.e., $Cycles\_C_i \subseteq Cycles\_C_j$ or $Cycles\_C_j \subseteq Cycles\_C_i$.

(2) The set $Cycles\_C_i$ is not a subset of the set $Cycles\_C_j$, or vice versa, but these two sets contain some equal elements. That is, $Cycles\_C_i \not\subset Cycles\_C_j$ and $Cycles\_C_j \not\subset Cycles\_C_i$ and $Cycles\_C_i \cap Cycles\_C_j \neq \emptyset$.

(3) The intersection of two sets $Cycles\_C_i$ and $Cycles\_C_j$ is empty set, i.e., $Cycles\_C_i \cap Cycles\_C_j = \emptyset$.

For the first situation, if $Cycles\_C_i \subseteq Cycles\_C_j$, suppose that cycle $c_j = C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n \rightarrow C_j \rightarrow C_1$ ($1 \leq n \leq m$, m is the number of classes) belongs to $Cycles\_C_j$ whereas the cycle $c_i = C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n \rightarrow C_i \rightarrow C_1$ does not belong to $Cycles\_C_i$, which means the cycle c_j but not c_i exists in the program. Because the common path $p = C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n$ exists, the disappearance of cycle c_i is caused by at least one of two class dependencies ($C_n \rightarrow C_i$ or $C_i \rightarrow C_1$) does not exist. Hence, class C_n depends on class C_j but not class C_i, or class C_1 is depended by the class C_j but not class C_i, or both two cases hold on. No matter which cases, this contradicts the assumption that class C_i and C_j have identical class dependence, so our assumption is false. In other words, the set $Cycles\_C_i$ is not a subset of the set $Cycles\_C_j$. Similarly, $Cycles\_C_j \subseteq Cycles\_C_i$ does not hold on.

For the other two situations, we can make the same supposition that cycle $c_j = C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n \rightarrow C_j \rightarrow C_1$ belongs to $Cycles\_C_j$ whereas the cycle $c_i = C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n \rightarrow C_i \rightarrow C_1$ does not exist. Based on the similar

inference, we can conclude that these two situations do not hold on, either. Therefore, all three situations are false, i.e., $Cycles\_C_i \neq Cycles\_C_j$ is false. In other words, the set of cycles containing class C_i is the same as the set of cycles containing class C_j, i.e., $Cycles\_C_i = Cycles\_C_j$.

Definition 2 (*Symmetric Classes*)**.** Two classes C_i and C_j, are said to be symmetric if they satisfy the following three conditions, (1) they depend on the same set of classes, i.e., $Target\_C_i = Target\_C_j$, (2) their dependent classes are equal, i.e., $Source\_C_i = Source\_C_j$, (3) for any pair of class dependencies, such as $C_1 \rightarrow C_i$ and $C_1 \rightarrow C_j$ (or $C_i \rightarrow C_1$ and $C_j \rightarrow C_1$), their attribute coupling and method coupling are identical, respectively.

Proposition 2. *Given a stable CITO generation algorithm, if class C_i and C_j are symmetric classes, then the generated test orders of class C_i and C_j are equal, i.e.,$Order\_C_i = Order\_C_j$.*

Proof. Because class C_i and C_j are symmetric classes, if class C_i is not involved in any cycles, then class C_j is not involved in any cycles, either. For a stable CITO generation algorithm, if there are no cycles, a class is integrated once all classes upon which it depends have been integrated and tested. All classes on which class C_i and C_j depend are the same, i.e., set $Target\_C_i = Target\_C_j$, therefore, in this case, the generated test orders of class C_i and C_j are equal. If class C_i is involved in cycles, according to Proposition 1, class C_j is also involved in the same cycles, i.e., $Cycles\_C_i = Cycles\_C_j$. For any cycle $c_i = C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n \rightarrow C_i \rightarrow C_1$ ($1 \leq n \leq m$, m is the number of classes) in $Cycles\_C_i$, there is a corresponding cycle $c_j = C_1 \rightarrow C_2 \rightarrow \ldots \rightarrow C_n \rightarrow C_j \rightarrow C_1$ in $Cycles\_C_j$. All Class dependencies are the same except for $C_n \rightarrow C_i$ ($C_i \rightarrow C_1$) and $C_n \rightarrow C_j$ ($C_j \rightarrow C_1$). Because the pair of class dependencies $C_n \rightarrow C_i$ ($C_i \rightarrow C_1$) and $C_n \rightarrow C_j$ ($C_j \rightarrow C_1$) have the same value of attribute coupling and method coupling, respectively, these two cycles are identical except for two different notations C_i and C_j. Correspondingly, solutions to break cycles in $Cycles\_C_i$ and $Cycles\_C_j$ are equal. Therefore, after removing some class dependencies, all classes on which class C_i and C_j depend are still the same. The generated test orders of class C_i and C_j are equal.

Similar-class combination is inspired by the graph reduction operation proposed by Orenstein et al. [9] for FVS identification problem. In theory, similar-class combination based on SCs cannot degrade the performance of any cycle-breaking technique, but in some cases, it can help to lower the stubbing cost of the generated CITOs. In contrast, similar-class combination based on ICD can both increase and lower the stubbing cost of the generated CITO depending on the specific case.

For the best case, similar-class combination can lower the stubbing cost of the generated CITO. For the example in Fig. 3 introduced in Section 2.2 Motivation, the cycle-breaking algorithms SCplx and CWR can generate the optimal solution for the reduced program, but these two algorithms cannot generate the optimal CITO for the original sample program. In the original program, the dependency $B \rightarrow C$ is involved in two cycles with the stubbing complexity (0.316), and the dependencies $C \rightarrow D$ and $E \rightarrow A$ have the lowest stubbing complexity (0.141 and 0.283) in these two cycles, separately. In this case, the dependency $B \rightarrow C$ cannot be selected by the two algorithms because it does not have the lowest stubbing cost or the greatest cycles–weight ratio. However, in the reduced program, the classes D and E are combined and consequently, the dependencies $C \rightarrow D$ and $E \rightarrow A$ do not exist. Two new class dependencies $C \rightarrow X$ and $X \rightarrow A$ are constructed, and their stubbing costs are higher (0.588 and 0.883) than that of $B \rightarrow C$. Therefore, the dependency $B \rightarrow C$ can be selected by the two algorithms.

Although ICD can identify a greater number of similar classes due to looser conditions, this advantage can be a drawback that increases the stubbing cost in the worst case. For example, suppose class C_i and class C_j have identical class dependence, indicating that these two classes are involved in the same cycles, that is, all of the classes in the corresponding cycles are equal, except for the classes C_i and C_j. Consider $Cycles\_C_i$ ($Cycles\_C_j$) to represent the set of cycles in which class C_i (C_j) is involved. Supposing that the class dependency $C \rightarrow C_i$ is the best choice to break all of the cycles in $Cycles\_C_i$, the corresponding class dependency $C \rightarrow C_j$ is guaranteed to break all of the cycles in $Cycles\_C_j$. However, this class dependency may not be the optimal solution for $Cycles\_C_j$ because of different stubbing complexity. The stubbing complexity of the class dependency $C \rightarrow C_j$ may be lower than, equal to, or even higher than that of the class dependency $C \rightarrow C_i$ due to the lack of restrictions on the coupling values of the class dependencies with identical class dependence. If the stubbing complexity of class dependency $C \rightarrow C_j$ is lower or equal, $C \rightarrow C_j$ is the best choice for $Cycles\_C_j$. Otherwise, there may exist another class dependency in $Cycles\_C_j$ with a stubbing complexity value between that of the class dependencies $C \rightarrow C_i$ and $C \rightarrow C_j$. In such a case, the class dependency $C \rightarrow C_j$ is no longer the optimal choice to break all of the cycles in $Cycles\_C_j$. We are unable to obtain a better solution when we regard similar classes as a whole and consider $Cycles\_C_i$ and $Cycles\_C_j$ together. This problem is described in greater detail in Section 4.3 with the aid of real applications.

Combining symmetric classes does not confront with the above-mentioned problem in similar-class identification by identical class dependence because symmetric classes specify the detailed constraints on the coupling values of similar classes, which ensures that the class dependencies related to these symmetric classes have the equal stubbing complexity. Therefore, for two symmetric classes C_i and C_j, the counterpart of the class dependency that is the best solution for $Cycles\_C_i$ must be the optimal result for $Cycles\_C_j$.

We apply these two properties, namely, identical class dependence and symmetric classes, in the proposed approach to identify similar classes. Similar classes can be combined and some redundant cycles containing such similar classes can be removed. These two properties mainly help to reduce the problem space for the existing CITO generation approaches, and theoretically, they have no effect on the stubbing cost if we use a stable cycle-breaking algorithm. However, a moderately large number of classes or cycles is an important reason for the poor effectiveness of the existing approaches, especially when these approaches are applied to a program with enormous class dependencies and complicated cycles. With heuristics, it would be easier to generate a better class test order for a program containing fewer classes and class dependencies. From this viewpoint, these two properties simplify the CITO generation problem for the original program, which helps to generate class test orders with a lower stubbing cost.

Algorithm 1 is the similar-class combination algorithm. Given a pair of classes C_i and C_j, we first calculate the source and target sets for them (lines 1–3). Then, based on the proposed properties, we identify whether these classes are similar (line 4). If they are similar, a new class C is created to replace them (line 5). The related class dependencies are constructed for the new class C (lines 6–7), and the stubbing complexity is calculated for these new class dependencies (lines 13–20). Finally, the sets of classes and cycles are updated (lines 9–10).

For simplicity, we consider only two classes in the algorithm, but in practice, multiple similar classes can be identified by using the proposed properties. Supposing that a new class C represents the combination of n similar classes, the set $sim(C)$ represents the set of these similar classes $C_1, C_2, \ldots, C_n$. For the class C_i in the set $sim(C)$, a new class dependency $S \rightarrow C$ is formed for each class S in $Source\_C_i$. Similarly, a new class dependency $C \rightarrow T$ is formed for each class T in $Target\_C_i$. The stubbing complexity of the new class dependencies is calculated using Eqs. (3) and (4), and it is the sum of stubbing complexities of the corresponding original class dependencies. After adding the new class C and its class dependencies into the program, the original similar classes and their related class dependencies are removed from the program.

Algorithm 1 Similar Class Combination

Input: a set of all classes $Classes$
a set of all cycles $Cycles$
Output: a set of combined classes $RClasses$
a set of reduced cycles $RCycles$
1: **for** each pair of classes C_i and C_j in $Classes$ **do**
2: calculate $Target\_C_i$ and $Target\_C_j$
3: calculate $Source\_C_i$ and $Source\_C_j$
4: **if** classes C_i and C_j own identical class dependence or are symmetric classes **then**
5: create a new class C
6: $Target\_C \leftarrow Target\_C_i$
7: $Source\_C \leftarrow Source\_C_i$
8: calculateSCplx()
9: $RClasses \leftarrow$ updateClasses(C_i, C_j, C)
10: $RCycles \leftarrow$ updateCycles(C_i, C_j, C)
11: **end if**
12: **end for**

13: **function** calculateSCplx()
14: **for** each class C_t in $Target(C)$ **do**
15: $SCplx(C \rightarrow C_t) = SCplx(C_i \rightarrow C_t) + SCplx(C_j \rightarrow C_t)$
16: **end for**
17: **for** each class C_s in $Source(C)$ **do**
18: $SCplx(C_s \rightarrow C) = SCplx(C_s \rightarrow C_i) + SCplx(C_s \rightarrow C_j)$
19: **end for**
20: **end function**

$$SCplx(S, C) = \sum_{C_i \in sim(C)} SCplx(S, C_i) \tag{3}$$

$$SCplx(C, T) = \sum_{C_i \in sim(C)} SCplx(C_i, T) \tag{4}$$

3.2. Breaking cycles in reduced programs

As discussed in Section 3.1, original programs can be reduced by means of similar-class combination based on the proposed properties. For such a reduced program that contains fewer classes and cycles, we construct an ORD, where each node represents a class and each edge represents the corresponding class dependency. Therefore, the CITO generation problem is transformed into the problem of traversing nodes in the ORD. The cycles in the diagram must be broken first, so that nodes can be visited by reverse topological sorting when no cycles exist, and consequently, the CITO is generated based on the node visitation order.

The cycles can be broken by removing the class dependencies. For each removed class dependency, a test stub is constructed. To minimize the stubbing cost of these test stubs, the most appropriate class dependency should be selected for removal. If the cycles among the class dependencies are independent and do not share common paths, the optimal solution is to remove the class dependency with the least stubbing complexity in each cycle. However, the existence of overlapping paths complexifies the class test order generation process.

Definition 3 (*Overlapping Paths*)**.** Supposing that c_A and c_B are two cycles in the object relation diagram for a given program. c_A is composed by m unique classes $c_A = C_{A1} \rightarrow C_{A2} \rightarrow \ldots \rightarrow C_{Am} \rightarrow C_{A1}$, and similarly, c_B consists of n unique classes $c_B = C_{B1} \rightarrow C_{B2} \rightarrow \ldots \rightarrow C_{Bn} \rightarrow C_{B1}$. For a path $P_A = C_{Ai} \rightarrow C_{A(i+1)} \rightarrow \ldots \rightarrow C_{Aj}$ $(1 \le i < j \le m)$ in c_A, if there exists a path in c_B, i.e., $P_B = C_{Bi} \rightarrow C_{B(i+1)} \rightarrow \ldots \rightarrow C_{Bj}$, with the same length as that of P_A, and the nodes in the same position are identical, i.e., $C_{Ai} = C_{Bi}$, $C_{A(i+1)} = C_{B(i+1)}$ and so on. Thus, c_A and c_B share common edges, P_A and P_B are overlapping paths.

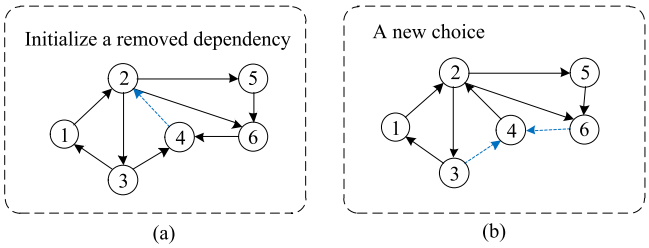


Fig. 5. An example of cycle breaking. A new choice (removing edges 3→4 and 6→4) with lower stubbing cost replaces the initially removed edge 4→2.

The overall stubbing cost of integration testing is calculated as the sum of the stubbing complexity of each class dependency, which can be affected by two factors. One is the number of removed dependencies; as the number of removed dependencies increases, the stubbing cost likely increases. The other is the stubbing complexity of each removed class dependency; the overall stubbing cost is likely to be low if all of the dependencies with relatively low stubbing complexity are removed. The overlapping paths in cycles tend to create a conflict between these two factors of the overall stubbing cost. This is also the reason why the existing cycle-breaking strategies do not yield satisfactory results when faced with overlapping cycles, as we have explained in Section 2.2. Two choices are presented when we deal with multiple overlapping cycles. Removing the common edges that are involved in multiple cycles requires fewer test stubs but the overall stubbing cost may increase because the construction of each test stub can be a complex process. Meanwhile, deleting the edge with the least stubbing complexity cannot guarantee an overall minimal stubbing cost because a greater number of edges may be removed than that with the former option.

We propose a new cycle-breaking algorithm called ANS because the basic idea of the proposed algorithm is to perform a good Analysis of the relationship between the Number of removed dependencies and the related Stubbing complexity. Our algorithm first selects a class dependency that breaks multiple cycles. Due to the existence of overlapping paths, there exist alternative solutions for the initially removed class dependency to break these cycles. Therefore, based on the cycles broken by this initial class dependency, our algorithm searches for another dependency or a set of other dependencies that break the same cycles but with a lower stubbing cost. The idea behind this algorithm is similar to the concept of "exploitation" in evolutionary algorithms, which searches for solutions that have better performance in the neighborhood of the current solution [18].

Fig. 5 shows an example of cycle-breaking. In this example, the object relation diagram contains six nodes. We choose the edge 4→2 as the initially removed dependency in Fig. 5(a), which breaks three cycles in the diagram: (1) 2→3→4→2, (2) 2→6→4→2, (3) 2→5→6→4→2. A new choice with lower stubbing cost devised by our cycle-breaking algorithm are edges 3→4 and 6→4 in Fig. 5(b). The edge 3→4 is removed to break the cycle (1), and edge 6→4 is deleted to break the cycles (2) and (3). Besides, only the new generated choice with lower stubbing cost is reserved, others are discarded.

Algorithm 2 is the proposed cycle-breaking algorithm, which attempts to maintain a good analysis between the two factors for minimizing the stubbing cost: the number of removed dependencies and the related stubbing complexity. All of the cycles in the program and all of the edges that are involved in the cycles constitute the input, and the algorithm generates a set of removed edges as its output. The algorithm begins with the initialization procedure (line 1), where the output set $RemEdges$ is created. Then, the edge that is involved in the greatest number of cycles is extracted (line 3) as the initially removed edge e. The stubbing complexity of edge e is calculated (line 4), and all of the cycles in which edge e is involved ($ECycles$) and all of the edges that are involved in these cycles ($EEdges$) are identified (lines 5–6). The

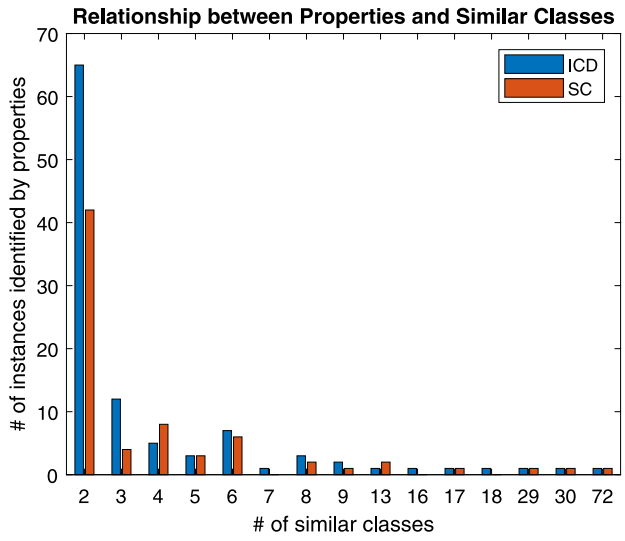


Fig. 6. Relationship between properties and similar classes.

algorithm then attempts to find a set of removed edges that can replace the initial edge, that is, breaking the same cycles or an equal number of cycles but with a lower stubbing cost (lines 7–8), which is executed by two functions, findFromCycles (lines 16–27) and findFromEdges (lines 28–41). The two sets $TempEdges1$ and $TempEdges2$ store the set of removed edges generated by findFromCycles and findFromEdges, respectively.

For the first function findFromCycles, we start from the cycles in which edge e is involved. For each cycle, edge e_t with the minimal stubbing complexity is obtained (line 18). If e_t is not equal to edge e and the overall stubbing cost after considering e_t is reduced (line 19), then e_t is added to the set $TempEdges1$, and consequently, all of the cycles in which e_t is involved are updated (lines 20–21). If the algorithm is unable to obtain better alternatives, it returns an empty set (line 23). After all of the cycles in set $ECycles$ have been processed, the final set $TempEdges1$ is obtained.

For the second function findFromEdges, we start from the edges that are involved in $ECycles$ and find a set of removed class dependencies that break the same number of cycles as the initial edge e. In these edges, the edge e_t with the maximal number of cycles is obtained (line 31). If removing e_t rather than e can reduce the overall stubbing cost, then e_t is added to the set $TempEdges2$ (lines 32–33). The number of remaining cycles is updated (line 34) as are the other edges in the set $EEdges$ (line 35). Similarly, an empty set is returned when no better alternatives exist (line 37). After the set $TempEdges2$ breaks the same number of cycles as the initial edge e, the execution of this function is completed.

Finally, the overall stubbing costs of these two sets obtained using the functions findFromCycles and findFromEdges are compared, and the better set is reserved (lines 9–10). Correspondingly, for the remaining edges, the cycles in which they are involved are updated (line 14). The above process is repeated until no cycles exist.

For the time complexity, assuming that the number of edges that are involved in cycles is E_I, and the number of cycles in the program is C. For each edge that is involved in the most cycles, we iterate all these cycles and their corresponding edges and then perform two functions. For these two functions, their main task is to compare the stubbing complexity of the initial edge with that of alternatives, whose time complexity is $O(C)$ and $O(E_I)$, respectively. The worst case is that we iterate all edges (E_I), therefore, the overall time complexity is $O(E_I^2 + C \cdot E_I)$.

Although both graph-based and search-based methods ignore these two factors, we did not solve this problem for search-based methods,

Algorithm 2 Cycle-Breaking ANS Algorithm

Input: a list of edges that are involved in cycles $AllEdges$
a list of cycles in the entire program $AllCycles$
Output: a set of removed edges $RemEdges$
1: $RemEdges \leftarrow \varnothing$
2: **for** $AllCycles$.size != 0 **do**
3: e = getMaxCyclesEdge($AllEdges$)
4: $scplx$ = getSCplx(e)
5: $ECycles$ = getCycles(e)
6: $EEdges$ = getEdges($ECycles$)
7: $TempEdges1 \leftarrow$ findFromCycles($ECycles$, $scplx$)
8: $TempEdges2 \leftarrow$ findFromEdges($EEdges$, $scplx$)
9: **if** $TempEdges1$.size != 0 || $TempEdges2$.size != 0 **then**
10: $RemEdges$.add(Compare($TempEdges1$, $TempEdges2$))
11: **else**
12: $RemEdges$.add(e)
13: **end if**
14: update($RemEdges$)
15: **end for**

16: **function** findFromCycles($ECycles$, $scplx$)
17: **for** each cycle c in $ECycles$ **do**
18: e_t = getMinSCplxEdge(c)
19: **if** $scplx$ > SCplx(e_t, $TempEdges1$) **then**
20: $TempEdges1$.add(e_t)
21: update(e_t)
22: **else**
23: $TempEdges1 \leftarrow \Phi$
24: **break**
25: **end if**
26: **end for**
27: **end function**

28: **function** findFromEdges($EEdges$, $scplx$)
29: num = getCyclesNum(e)
30: **for** num > 0 **do**
31: e_t = getMaxCyclesEdge($EEdges$)
32: **if** $scplx$ > SCplx(e_t, $TempEdges2$) **then**
33: $TempEdges2$.add(e_t)
34: num = num - getCyclesNum(e_t)
35: updateEdges(e_t)
36: **else**
37: $TempEdges2 \leftarrow \Phi$
38: **break**
39: **end if**
40: **end for**
41: **end function**

because these two factors (the number of test stubs and the corresponding stubbing complexity) can be considered only when dealing with an incomplete CITO. In graph-based methods, the CITO can be generated only after all of the cycles have been broken. Therefore, we can choose which class dependency is to be removed by considering these two factors. In contrast, in search-based methods, an integer represents a class, and a vector of such integers represents a CITO. All of the evolutionary operators are applied to full CITOs to generate an offspring. Therefore, the proposed measure for graph-based methods cannot be applied to search-based methods.

4. Experiments

We conducted several experiments to evaluate the performance of our approach. The information of used programs, experimental settings, and evaluation metrics are described in Section 4.1. Research

questions and the corresponding results and analyses are introduced in Sections 4.2 and 4.3, respectively. Section 4.4 presents the discussion. Section 4.5 discusses the threats to validity.

4.1. Experimental settings and evaluation metrics

We selected nine Java programs that are widely used in the existing CITO generation approaches to evaluate the performance of our approach. Table 2 summarizes these programs, including their sources, descriptions, and numbers of classes.

For these programs, the class dependencies and the related coupling information were obtained using a Java program analysis framework called Soot (http://www.sable.mcgill.ca/soot).

We used overall stubbing complexity to estimate the stubbing cost of the entire integration testing process. A class test order with the minimum value of OCplx was desired. The number of test stubs (Stubs) was also used as a supplementary metric. We considered that the fewer test stubs are required, the better is the performance of the CITO generation approach.

4.2. Research questions

Our approach first identifies similar classes by identifying the two properties pertaining to class dependence, namely, ICD and SC. Based on this notion of similar classes, some inter-class relationships can be effectively identified, and some redundant cycles containing these inter-class relationships can be removed. Thereafter, a reduced program that contains fewer classes and class dependencies is generated. For the reduced program, we adopt the proposed cycle-breaking algorithm (ANS) to minimize the stubbing cost. To evaluate the performance of the proposed approach, we designed experiments to answer the following three research questions. RQ1 and RQ2 pertain to the effectiveness of the similar-class combination and cycle-breaking algorithm, respectively, under the condition that the other factors remain unchanged. RQ3 assesses the performance of the proposed approach and explores the effects of similar-class combination on the performance of the state-of-the-art cycle-breaking techniques.

RQ1: What is the performance of similar-class combination in terms of reducing the problem space?

Before generating CITOs, we first analyze the effects of similar-class combination on reducing the problem space for graph-based and search-based methods. We collect two types of similar classes by identifying two properties, namely ICD and SC, as proposed in Section 3.1, and obtain two reduced programs by combining these similar classes. We count the number of classes, class dependencies, and cycles in the original program and the two reduced programs. The changes in the above statistics represent the effects of similar-class combination on minimizing the program scale, such as reduction of the number of classes and class dependencies.

RQ2: What is the performance of the proposed cycle-breaking algorithm (ANS) in terms of minimizing the stubbing cost?

To answer this question, we evaluate whether the proposed cycle-breaking algorithm (ANS) reduces the stubbing cost of the original program compared with the state-of-the-art cycle-breaking techniques. The three cycle-breaking strategies mentioned in Section 2.2 are used as competitors to validate the effectiveness of the proposed algorithm. For the ORD constructed for the original program, these strategies use the following three rules to remove the edges in the diagram until all of the cycles are broken.

- Number of Cycles (NC) removes the edge that is involved in the greatest number of cycles, which is adopted by Briand et al. [7].
- Stubbing Complexity (SCplx) removes the edge with the minimal stubbing complexity, which is applied by Hashim et al. [11].
- Cycles–Weight Ratio (CWR) removes the edge with the highest cycles–weight ratio, such as the methods proposed by Bansal et al. [13] and Abdurazik et al. [14]. The idea behind this rule is that removing such an edge may break the most cycles with minimal stubbing cost. In this experiment, weight for each edge is the stubbing complexity for the corresponding class dependency.

After eliminating all of the cycles, test stubs are constructed for the removed class dependencies. The overall stubbing complexity and the number of test stubs are counted and compared.

RQ3: Will similar-class combination affect the performance of the cycle-breaking techniques?

We evaluate whether similar-class combination affects the performance of the cycle-breaking techniques. We apply the cycle-breaking algorithms mentioned in RQ2 and the proposed algorithm to the reduced programs obtained by similar-class combination and compare the results.

4.3. Results and analyses

RQ1: The performance of similar-class combination in reducing the problem space.

We recorded the set of similar classes identified based on the two proposed properties, namely ICD and SC. Table 3 lists the number of records for similar classes that were identified based on ICD and SC for each program. The statistics show that all programs contained similar classes that were identified by the two proposed properties, except for the program *deos*. For the large-scale programs with more than 100 classes, we identified greater numbers of similar classes than those for small-scale programs, such as programs *daisy*, *email_spl*, and *notepad_spl*. In general, SC identified fewer similar classes due to its stricter conditions compared with the property of ICD.

We also counted the number of similar classes in each record that were identified by the two properties ICD and SC for all programs. Fig. 6 shows the relationship between the proposed properties and similar classes. The value on the y-axis indicates the number of sets with similar classes whose sizes, indicated on the x-axis, are obtained using the corresponding property. Among the sets obtained based on ICD and SC, in 61.9% (65/105) and 58.3% (42/72), respectively, two similar classes were identified. Most of the sets of similar classes contained no more than 10 similar classes, and the largest set contained 72 similar classes.

Table 4 summarizes a comparison of the programs obtained by means of similar-class combination based on the two properties ICD and SC in terms of the number of classes, number of class dependencies (Deps), number of class dependencies that are involved in cycles (OverDeps), and number of cycles. The columns “Original”, “ICD”, and “SC” present the statistics of the original programs and the programs obtained by means of similar-class combination based on ICD and SC, respectively. The percentage values describe the degree of the reduction in these statistics. Most programs contained fewer classes and class dependencies after similar-class combination, except for the program *deos*. For six programs (*email_spl*, *GPL_spl*, *JHotDraw*, *log4j3*, *MyBatis* and *notepad_spl*), more than 10% reductions in the number of classes and more than 6% reductions in the number of dependencies were achieved. For four programs (*email_spl*, *GPL_spl*, *MyBatis* and *notepad_spl*), more than 20% reductions in the number of cycles were achieved. The ICD property identified a greater number of similar classes, and it achieved the highest reductions for the *notepad_spl* program: approximately 55.38% (1-29/65), 62.41% (1-53/141), 70.49% (1-36/122), and 86.78% (1-30/227) in the numbers of classes, dependencies, OverDeps, and cycles, respectively.

To highlight the differences between the original programs and the reduced programs more intuitively, we drew the ORDs for the strongly connected components that contained similar classes. Fig. 7 shows a comparison of two ORDs generated for the program *notepad_spl*.

Table 4
Comparison of programs obtained by combining similar classes based on two properties — columns 'Original', 'ICD' and 'SC' present the statistics in the original programs, programs obtained by combining similar classes based on ICD and SC, respectively. Percentage below describes the degree of the reduction in these items compared with the original programs.

Programs	# of classes			# of Deps			# of OverDeps			# of cycles		
	Original	ICD	SC	Original	ICD	SC	Original	ICD	SC	Original	ICD	SC
daisy	23	22 4.35%	22 4.35%	36	35 2.78%	35 2.78%	9	9 0.00%	9 0.00%	4	4 0.00%	4 0.00%
email_spl	39	29 25.64%	31 20.51%	61	49 19.67%	53 13.11%	28	23 17.86%	25 10.71%	38	26 31.58%	30 21.05%
GPL_spl	178	103 42.13%	114 35.96%	260	150 42.31%	167 35.77%	104	68 34.62%	80 23.08%	144	88 38.89%	103 28.47%
JHotDraw	411	357 13.14%	366 10.95%	1680	1417 15.65%	1458 13.21%	199	180 9.55%	186 6.53%	225	204 9.33%	210 6.67%
jmeter	372	340 8.60%	352 5.38%	1252	1204 3.83%	1238 1.12%	147	137 6.80%	145 1.36%	729	721 1.10%	728 0.14%
log4j3	261	211 19.16%	226 13.41%	784	662 15.56%	705 10.08%	242	162 33.06%	186 23.14%	2173	2053 5.52%	2136 1.70%
MyBatis	428	302 29.44%	327 23.60%	1211	1058 12.63%	1131 6.61%	389	327 15.94%	351 9.77%	37,882	14,790 60.96%	23,861 37.01%
notepad_spl	65	29 55.38%	31 52.31%	141	53 62.41%	55 60.99%	122	36 70.49%	36 70.49%	227	30 86.78%	30 86.78%

Table 5
Comparison of results on original programs for our cycle-breaking algorithm (ANS) and three competitors (NC, SCplx and CWR). The best values are highlighted in bold.

Programs	Methods	OCplx			Stubs		
		Range	Mean	*p*-value	Range	Mean	*p*-value
daisy	NC	[0.129–0.482]	0.319	1.666e−09	3	**3**	–
	SCplx	0.161	0.161	–	4	4	–
	CWR	[0.129–0.161]	0.145	9.651e−06	[3–4]	3.5	9.651e−06
	ANS	0.129	**0.129**	–	3	**3**	–
deos	NC	[1.547–3.180]	2.595	1.191e−12	[11–13]	12.333	0.01548
	SCplx	2.695	2.695	–	27	27	–
	CWR	1.351	1.351	–	14	14	–
	ANS	1.257	**1.257**	–	12	**12**	–
email_spl	NC	[0.957–1.168]	1.075	1.111e−12	5	**5**	–
	SCplx	[0.972–1.061]	1.008	1.052e−12	[6–11]	8.033	1.052e−12
	CWR	0.902	**0.902**	–	5	**5**	–
	ANS	0.902	**0.902**	–	5	**5**	–
GPL_spl	NC	[2.488–3.959]	3.060	1.212e−12	42	**42**	–
	SCplx	[2.950–2.981]	2.967	4.455e−13	[52–53]	52.567	4.455e−13
	CWR	2.582	2.582	–	49	49	–
	ANS	2.244	**2.244**	–	42	**42**	–
JHotDraw	NC	[1.599–2.328]	1.979	1.212e−12	90	**90**	–
	SCplx	[0.977–1.004]	0.988	8.691e−13	[94–98]	95.700	8.691e−13
	CWR	0.963	**0.963**	–	92	92	–
	ANS	0.963	**0.963**	–	91	91	–
jmeter	NC	[1.998–2.694]	2.278	1.212e−12	39	**39**	–
	SCplx	[1.467–1.547]	1.513	1.132e−12	[55–60]	57.833	9.807e−13
	CWR	[1.305–1.335]	1.320	4.696e−13	[42–43]	42.500	4.696e−13
	ANS	1.284	**1.284**	–	40	40	–
log4j3	NC	[2.433–3.065]	2.816	1.212e−12	90	**90**	–
	SCplx	[2.162–2.425]	2.297	1.199e−12	[103–120]	111.833	1.167e−12
	CWR	1.972	1.972	–	96	96	–
	ANS	1.965	**1.965**	–	91	91	–
MyBatis	NC	[1.603–2.580]	2.103	1.212e−12	[58–59]	58.567	4.455e−13
	SCplx	[2.086–2.271]	2.171	1.210e−12	[119–132]	124.167	1.145e−12
	CWR	1.201	**1.201**	–	[60–62]	61.800	6.115e−14
	ANS	1.209	1.209	–	57	**57**	–
notepad_spl	NC	[2.489–3.480]	3.031	1.211e−12	43	**43**	–
	SCplx	[1.249–1.478]	1.347	1.132e−12	[56–68]	61.167	1.132e−12
	CWR	1.019	1.019	–	44	44	–
	ANS	1.000	**1.000**	–	43	**43**	–

Due to space limitations in this paper, the diagrams of the other programs can be accessed on GitHub (https://github.com/miazhang9/CITO_SCC). Fig. 7(a) shows the ORD for the original program, where the value in each node indicates the class number, and the directed edge represents the dependency between two classes. The ORDs of the two reduced programs obtained using ICD and SC are identical, as shown in Fig. 7(b). Two types of similar classes that are shown in Fig. 7 were identified by both properties, and the class numbers of these classes are shown as follows:

Table 6
Comparison of results on programs after combining similar classes by property ICD. Values that are different from those in Table 5 are highlighted in bold.

Programs	Methods	OCplx			Stubs		
		Range	Mean	*p*-value	Range	Mean	*p*-value
daisy	NC-ICD	[0.129–0.482]	**0.332**	5.186e−09	3	3	–
	SCplx-ICD	0.161	0.161	–	4	4	–
	CWR-ICD	[0.129–0.161]	**0.144**	2.364e−05	[3–4]	**3.467**	2.364e−05
	ANS-ICD	0.129	0.129	–	3	3	–
email_spl	NC-ICD	[**0.902**–1.168]	**1.061**	1.583e−11	5	5	–
	SCplx-ICD	[0.972–**1.026**]	**1.002**	9.338e−13	[6–**9**]	**7.667**	9.338e−13
	CWR-ICD	0.902	0.902	–	5	5	–
	ANS-ICD	0.902	0.902	–	5	5	–
GPL_spl	NC-ICD	[**2.572–3.849**]	**3.076**	1.212e−12	42	42	–
	SCplx-ICD	[**2.960–2.991**]	**2.976**	9.338e−13	[52–53]	**52.5**	4.696e−13
	CWR-ICD	**2.592**	**2.592**	–	49	49	–
	ANS-ICD	**2.254**	**2.254**	–	42	42	–
JHotDraw	NC-ICD	[**1.439–2.207**]	**1.904**	1.211e−12	90	90	–
	SCplx-ICD	[**1.076–1.097**]	**1.087**	7.718e−13	[94–**97**]	**95.600**	7.718e−13
	CWR-ICD	**1.063**	**1.063**	–	92	92	–
	ANS-ICD	**1.063**	**1.063**	–	91	91	–
jmeter	NC-ICD	[**1.890–2.738**]	**2.295**	1.212e−12	39	39	–
	SCplx-ICD	[1.467–**1.588**]	**1.524**	1.150e−12	[55–**62**]	**58.333**	1.087e−12
	CWR-ICD	[1.305–**1.345**]	**1.322**	5.634e−13	[42–**44**]	**42.600**	5.634e−13
	ANS-ICD	1.284	1.284	–	40	40	–
log4j3	NC-ICD	[**2.405–3.186**]	**2.828**	1.212e−12	90	90	–
	SCplx-ICD	[**2.145–2.381**]	**2.257**	1.206e−12	[**102–117**]	**109.3**	1.144e−12
	CWR-ICD	**1.929**	**1.929**	–	**94**	**94**	–
	ANS-ICD	1.965	1.965	–	91	91	–
MyBatis	NC-ICD	[**1.574–2.610**]	**2.034**	1.212e−12	**58**	**58**	–
	SCplx-ICD	[**1.709–1.920**]	**1.802**	1.210e−12	[**101–114**]	**106.7**	1.162e−12
	CWR-ICD	1.201	1.201	–	[60–62]	**61.533**	1.969e−13
	ANS-ICD	1.209	1.209	–	57	57	–
notepad_spl	NC-ICD	[**1.164–1.846**]	**1.462**	1.211e−12	**42**	**42**	–
	SCplx-ICD	[**1.038–1.650**]	**1.320**	1.068e−12	[**45–77**]	**59.733**	1.068e−12
	CWR-ICD	1.019	1.019	–	44	44	–
	ANS-ICD	**0.981**	**0.981**	–	**42**	**42**	–

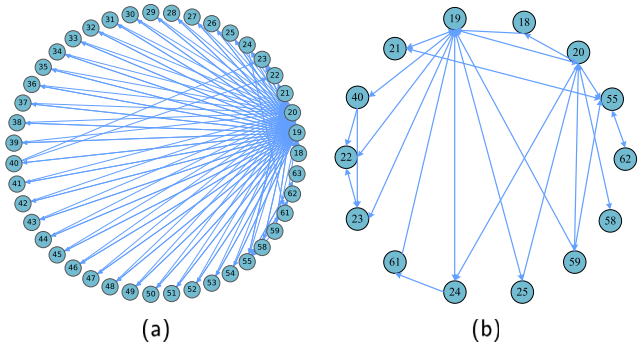


Fig. 7. Comparison of object relation diagram for notepad_spl: (a) Original program (b) Reduced program.

- 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54
- 62, 63

As shown in Fig. 7(a), the strongly connected component extracted from the original program contains 43 classes and 120 class dependencies. While in the reduced program, classes 62 and 63 are combined as an integral. Similarly, class 25 and other 28 classes are treated holistically. Therefore, we only need to tackle 14 classes and 34 class dependencies in the reduced program as shown in Fig. 7(b). The new program contains only 30 cycles compared with 227 cycles in the original program.

A reduction in the number of cycles improves the efficiency of the graph-based approaches because these approaches aim to break all of the cycles in the program. For search-based approaches, a reduction in the number of classes is helpful due to the encoding strategy for CITOs, as mentioned in Section 1. The size of the search space for search-based methods is $N!$ (N factorial) for a program containing N classes. The fewer the classes, the smaller is the search space. Hence, the conventional CITO generation approaches can benefit from similar-class combination based on the proposed properties.

RQ2: Performance of proposed cycle-breaking algorithm (ANS) on minimizing stubbing cost.

We evaluated the performance of the proposed cycle-breaking algorithm and compared it with the performance of the three other typical cycle-breaking strategies studied in Section 2.2. Each strategy was executed 30 times for each program. The non-deterministic results obtained using the other three methods were subjected to the Wilcoxon rank-sum test [19] to evaluate whether there exist statistically significant differences between the OCplx and Stubs obtained by our algorithm and those obtained using the other algorithms. The process of evaluating the existence of differences is consistent with the literature [7]. The ranges, means and *p*-values of OCplx and Stubs are listed in Table 5, where the minimal mean values are presented in boldface.

In terms of the range of OCplx, NC produced a distribution of widely varying results across the 30 executions for all programs because multiple dependencies were involved in the same number of cycles in the later stage of the cycle-breaking algorithm. The range of OCplx obtained using SCplx was also wide for some programs, such as *jmeter* and *log4j3*, but the range was narrower than that of NC. These results indicate that the programs might contain multiple dependencies with the same minimal stubbing complexity, but the probability is relatively low because the same stubbing complexities require both the same attribute coupling and method coupling for two different class dependencies. In contrast, CWR and the proposed algorithm are

Table 7
Comparison of results on programs after combining similar classes by property SC. Values that are different from those in Table 5 are highlighted in bold.

Programs	Methods	OCplx			Stubs		
		Range	Mean	*p*-value	Range	Mean	*p*-value
daisy	NC-SC	[0.129–0.482]	**0.315**	1.591e−08	3	3	–
	SCplx-SC	0.161	0.161	–	4	4	–
	CWR-SC	[0.129–0.161]	**0.148**	5.189e−07	[3–4]	**3.6**	5.189e−07
	ANS-SC	0.129	0.129	–	3	3	–
email_spl	NC-SC	[**0.902–1.150**]	**1.054**	4.257e−12	5	5	–
	SCplx-SC	[0.972–**1.026**]	**1.000**	8.727e−13	[6–**9**]	**7.533**	8.727e−13
	CWR-SC	0.902	0.902	–	5	5	–
	ANS-SC	0.902	0.902	–	5	5	–
GPL_spl	NC-SC	[**2.467–3.795**]	**3.000**	1.212e−12	42	42	–
	SCplx-SC	[2.950–2.981]	**2.969**	3.798e−13	[52–53]	**52.633**	3.798e−13
	CWR-SC	2.582	2.582	–	49	49	–
	ANS-SC	2.244	2.244	–	42	42	–
JHotDraw	NC-SC	[**1.681–2.245**]	**1.985**	1.212e−12	90	90	–
	SCplx-SC	[0.977–**0.997**]	0.988	8.815e−13	[94–**97**]	**95.633**	8.815e−13
	CWR-SC	0.963	0.963	–	92	92	–
	ANS-SC	0.963	0.963	–	91	91	–
jmeter	NC-SC	[**1.946–3.004**]	**2.349**	1.212e−12	39	39	–
	SCplx-SC	[1.467–**1.578**]	**1.526**	1.151e−12	[55–**61**]	**58.333**	1.015e−12
	CWR-SC	[1.305–**1.345**]	1.320	5.669e−13	[42–**44**]	**42.567**	5.669e−13
	ANS-SC	1.284	1.284	–	40	40	–
log4j3	NC-SC	[**2.432–3.276**]	**2.790**	1.212e−12	90	90	–
	SCplx-SC	[**2.145–2.414**]	**2.293**	1.212e−12	[**102–119**]	**111.567**	1.179e−12
	CWR-SC	1.972	1.972	–	96	96	–
	ANS-SC	1.965	1.965	–	91	91	–
MyBatis	NC-SC	[**1.602–2.471**]	**2.023**	1.212e−12	[58–59]	**58.633**	3.798e−13
	SCplx-SC	[**1.743–1.948**]	**1.839**	1.209e−12	[**102–116**]	**109.433**	1.171e−12
	CWR-SC	1.201	1.201	–	[60–62]	**61.533**	1.969e−13
	ANS-SC	1.209	1.209	–	57	57	–
notepad_spl	NC-SC	[**1.060–1.618**]	**1.368**	1.208e−12	**42**	**42**	–
	SCplx-SC	[**1.019–1.650**]	**1.279**	1.069e−12	[**44–77**]	**57.567**	1.069e−12
	CWR-SC	1.019	1.019	–	44	44	–
	ANS-SC	**0.981**	**0.981**	–	**42**	**42**	–

Table 8
Class dependencies causing better result of CWR-ICD for program log4j3.

Class Deps	# of cycles	Stubbing complexity	Cycles–weight ratio
36→16	2	0.011	181.818
16→18	8	0.098	81.633
38→16	2	0.032	62.500

more deterministic because they rarely make arbitrary decisions when removing class dependencies.

Our cycle-breaking algorithm obtained the minimal mean OCplx values for eight of nine programs, and the exception was the program *MyBatis*. Both CWR and the proposed algorithm achieved the minimal mean OCplx value for the programs *email_spl* and *JHotDraw*. For the remaining programs, the OCplx values obtained using the proposed algorithm were lower than the values obtained using the other algorithms, and the reductions achieved using the proposed algorithm ranged from 0.35% (1–1.965/1.972) to 13.09% (1–2.244/2.582) reduction. Overall, all of the OCplx values obtained using the proposed algorithm are comparable to if not better than the best results obtained using the other approaches. For the program *MyBatis*, our algorithm performed slightly worse than CWR because of the excessive removal of class dependencies due to the initialization. To accelerate cycle-breaking, we started from an initially removed edge that was involved in the greatest number of cycles. However, such a choice may have led to a sub-optimal result, such as the result obtained for the program *MyBatis*.

Although NC created the fewest test stubs for seven programs (except for the programs *deos* and *MyBatis*), its stubbing cost was unsatisfactory. Thus, considering only the number of test stubs is inadequate for minimizing the stubbing cost. The proposed algorithm follows NC in terms of Stubs, meaning that it comprehensively considers the number of test stubs and the corresponding stubbing complexity when determining the overall stubbing cost. In addition, the *p*-values below 0.05 indicate that the differences between OCplx and Stub values obtained using the proposed method and the other methods are statistically significant.

RQ3: Effects of similar-class combination on the cycle-breaking techniques.

To answer RQ3, we evaluate the effects of similar-class combination on the performance of the cycle-breaking techniques, and this question includes two research sub-questions:

RQ3.1 What is the performance of the proposed cycle-breaking algorithm in minimizing the stubbing cost when applying similar-class combination?

RQ3.2 Will similar-class combination affect the performance of other cycle-breaking techniques?

Tables 6 and 7 present the OCplx and Stubs obtained using the three competitors and the proposed cycle-breaking algorithm for the reduced programs, that is, the programs obtained by means of similar-class combination based on ICD and SC, respectively. The results obtained for all of the programs, except for *deos* (which does not have any instances of similar classes identified based on the two properties), are compared. As in the case of RQ2, each method was executed 30 times for each program because some competitors are non-deterministic. The Wilcoxon rank-sum test was performed to investigate whether the differences between the results obtained using the proposed method and those obtained using the existing methods were statistically significant.

For RQ3.1, as shown in Table 6, the proposed approach achieved the minimal stubbing cost for six of the eight programs (except for *log4j3* and *MyBatis*). The results shown in Table 7 are identical to our findings for RQ2: the proposed approach outperformed the competitor approaches for all programs, except *MyBatis*.

Table 9
P-values for methods performing on programs before and after combining similar classes.

Programs	OCplx				Stubs	
	NC-ICD vs. NC	NC-SC vs. NC	SCplx-ICD vs. SCplx	SCplx-SC vs. SCplx	SCplx-ICD vs. SCplx	SCplx-SC vs. SCplx
daisy	0.808	0.705	–	–	–	–
email_spl	0.498	0.352	0.467	0.278	0.467	0.278
GPL_spl	0.741	0.495	**0.003**	0.607	0.614	0.607
JHotDraw	0.149	0.878	**1.644e−11**	0.926	0.846	0.926
jmeter	0.687	0.248	0.297	0.056	0.334	0.175
log4j3	0.623	0.406	0.052	0.994	0.053	1.000
MyBatis	0.382	0.279	**3.012e−11**	**3.01e−11**	**2.8e−11**	**2.816e−11**
notepad_spl	**3.016e−11**	**3.01e−11**	0.662	0.184	0.662	0.184

For *MyBatis*, the proposed approach yielded the same OCplx and Stubs values as those without similar-class combination, as well as the same as those yielded by CWR. However, the proposed approach performed slightly worse than CWR due to the extra removed class dependencies, as it did in the case of RQ2.

For the program *log4j3*, CWR-ICD reduced 0.036 (1.965–1.929) on OCplx, which is better than the performance of the original CWR (1.972) and that of the proposed approach (1.965). CWR-ICD yielded a better choice because it established the association between cycles by means of ICD. For example, CWR-ICD removed the class dependency 16→18 instead of 36→16 and 38→16, as selected by CWR. Table 8 shows the information of these three class dependencies; the class dependency 36→16 has a higher cycles–weight ratio than the other class dependencies. Therefore, the class dependency 36→16 was first removed by CWR. Then, the cycles–weight ratio was updated for the remaining class dependencies. The class dependency 16→18 was involved in eight cycles, where two cycles also contained the class dependency 36→16. Therefore, its cycles–weight ratio was updated to 61.224 (6/0.098 = 61.224). The class dependency 38→16 was still involved in two cycles, so it yielded the highest cycles–weight ratio (62.500). Therefore, CWR removed the class dependency 38→16. However, classes 36 and 38 exhibited ICD. In this case, CWR-ICD regarded classes 36 and 38 holistically, replaced them with a new class X, and calculated the cycles–weight ratio for the combined class dependency X→16 (2/(0.011 + 0.032) = 46.512). Correspondingly, the class dependency 16→18 was involved in six cycles, and its cycles–weight ratio was the highest (6/0.098 = 61.224). Although it seems that removing class dependency 16→18 led to the greatest increase in stubbing cost by far, CWR-ICD eventually yielded a better result.

In terms of Stubs, NC yielded the least number of test stubs in most cases, followed by the proposed approach. This finding is consistent with our findings for RQ2. Moreover, the p-values lower than 0.05 indicate that after similar-class combination, the differences between the results obtained using the competitors and those obtained using the proposed approach are statistically significant.

For RQ3.2, In Tables 6 and 7, values of OCplx and Stubs that are different from those in Table 5 are presented in boldface for the sake of readability, while the best values of OCplx and Stubs are presented in boldface in Table 5.

The results obtained using CWR and the proposed approach were deterministic for most of the programs. Therefore, the differences in the results obtained before and after similar-class combination are obvious. For the non-deterministic values of OCplx and Stubs, we conducted the Wilcoxon rank-sum test to estimate whether the results obtained by means of similar-class combination based on ICD and SC differed significantly from the original results. Table 9 presents the p-values of eight programs for NC and SCplx, respectively. The p-values below 0.05 indicate significant differences, and they are presented in boldface.

The p-values of Stubs for NC were not shown because for seven programs, NC, NC-ICD and NC-SC all obtained certain values. For *MyBatis*, the p-values for NC-ICD and NC-SC were 1.434e−06 and 0.607 (Stubs), respectively. It indicates that the number of test stubs generated by NC and NC-ICD has significant differences, but it does not hold for NC-SC.

Table 10
Programs for which generated CITOs are affected by similar class combination. The up arrow indicates higher OCplx (Stubs) compared with its original result, whereas the down arrow means lower OCplx (Stubs).

Methods	OCplx	Stubs
NC-ICD	notepad_spl(↓)	MyBatis(↓), notepad_spl(↓)
SCplx-ICD	GPL_spl(↑), JHotDraw(↑), MyBatis(↓)	MyBatis(↓)
CWR-ICD	GPL_spl(↑), JHotDraw(↑), log4j3(↓)	log4j3(↓)
ANS-ICD	GPL_spl(↑), JHotDraw(↑), notepad_spl(↓)	notepad_spl(↓)
NC-SC	notepad_spl(↓)	notepad_spl(↓)
SCplx-SC	MyBatis(↓)	MyBatis(↓)
ANS-SC	notepad_spl(↓)	notepad_spl(↓)

CWR, CWR-ICD, and CWR-SC all obtained deterministic values for most programs. With regard to the remaining programs, for *daisy*, the p-values for CWR-ICD and CWR-SC were 0.804 (OCplx) and 0.445 (Stubs). For *jmeter*, the p-values for CWR-ICD and CWR-SC were 0.6202 (both OCplx and Stubs) and 0.9068, respectively. For *MyBatis*, the p-values for CWR-ICD and CWR-SC were 0.173 (Stubs). It indicates that no statistically significant differences between the results from CWR-ICD (or CWR-SC) and CWR exist. Therefore, similar-class combination does not degrade the performance of CWR in terms of minimizing stubbing cost for programs *daisy*, *jmeter* and *MyBatis*.

By comparing the OCplx and Stubs obtained before and after similar-class combination, we found that combining SCs minimized the number of classes (as well as class dependencies) without degrading the performance of the cycle-breaking techniques in terms of minimizing stubbing cost for all programs. The other property ICD did not degrade the performance of the cycle-breaking techniques for most programs. We will discuss the other special cases in Section 4.4.

Overall, the proposed approach outperformed the competitors on most programs, and especially when combining SCs, it achieved the minimal stubbing cost on seven of the eight programs.

4.4. Discussion

In some special cases, similar-class combination based on the two proposed properties, namely ICD and SC, may affect the specific characteristics of the used programs. Correspondingly, different CITOs are generated by these cycle-breaking techniques when combining similar classes. We discuss such special cases in this section. We summarize the programs for which methods obtained significantly different OCplx and Stubs under similar-class combination based on the two properties and explain these differences.

Table 10 presents the programs for which the generated CITOs are affected by similar-class combination. The upward arrows between parentheses indicate that a approach obtained better results than the original version, whereas the downward arrows indicate lower OCplx or Stubs. The results show that similar-class combination based on a stricter property, that is, SC, not only did not affect the performance of the cycle-breaking techniques but also helped to reduce the stubbing cost for some programs (such as *notepad_spl* and *MyBatis*), which is consistent with the explanation given in Section 3.1. Although the

Cycle [152]: 20; 19; 20;	Cycle [162]: 20; 19; 34; 20;	Cycle [172]: 20; 19; 45; 20;
Cycle [153]: 20; 19; 25; 20;	Cycle [163]: 20; 19; 35; 20;	Cycle [173]: 20; 19; 46; 20;
Cycle [154]: 20; 19; 26; 20;	Cycle [164]: 20; 19; 36; 20;	Cycle [174]: 20; 19; 47; 20;
Cycle [155]: 20; 19; 27; 20;	Cycle [165]: 20; 19; 37; 20;	Cycle [175]: 20; 19; 48; 20;
Cycle [156]: 20; 19; 28; 20;	Cycle [166]: 20; 19; 38; 20;	Cycle [176]: 20; 19; 49; 20;
Cycle [157]: 20; 19; 29; 20;	Cycle [167]: 20; 19; 39; 20;	Cycle [177]: 20; 19; 50; 20;
Cycle [158]: 20; 19; 30; 20;	Cycle [168]: 20; 19; 41; 20;	Cycle [178]: 20; 19; 51; 20;
Cycle [159]: 20; 19; 31; 20;	Cycle [169]: 20; 19; 42; 20;	Cycle [179]: 20; 19; 52; 20;
Cycle [160]: 20; 19; 32; 20;	Cycle [170]: 20; 19; 43; 20;	Cycle [180]: 20; 19; 53; 20;
Cycle [161]: 20; 19; 33; 20;	Cycle [171]: 20; 19; 44; 20;	Cycle [181]: 20; 19; 54; 20;

Fig. 8. Cycles containing class dependency 20→19.

OCplx values for the programs *GPL_spl* and *JHotDraw* increased slightly for the three approaches when similar-class combination based on ICD was applied, this drawback disappeared when SCs were combined. This result can be ascribed to the distinct coupling values of these class dependencies for similar-class combination. For example, three classes (35, 36, and 38) in *GPL_spl* form four class dependencies: 35→36 and 38→36 (attribute coupling = 2, method coupling = 2, stubbing complexity = 0.072), 36→35 (attribute coupling = 0, method coupling = 3, stubbing complexity = 0.092) and 36→38 (attribute coupling = 0, method coupling = 2, stubbing complexity = 0.061). These class dependencies are involved in two cycles: (1) 35→36→35 and (2) 38→36→38. Classes 35 and 38 have ICD based on Proposition 1 rather than are SCs based on Proposition 2 because class dependency 36→35 and 36→38 have different coupling values. If classes 35 and 38 were combined, only one cycle existed. In the case of the NC approach, such combination of classes did not influence the final result because all class dependencies had the same number of cycles as before. While for the other three approaches, class dependencies 35→36 and 38→36 were removed due to lower stubbing complexity than that of class dependencies 36→35 and 36→38. However, the optimal choice was removing class dependencies 35→36 and 36→38, which can be obtained when these two classes were not combined. The same reason is true for the program *JHotDraw*.

For *notepad_spl*, both NC and the proposed approach achieved fewer OCplx and Stubs with similar-class combination based on ICD and SC. The class dependency involved in the greatest number of cycles was removed by NC, but this abundance of cycles can be reduced by the similar classes identified by ICD and SC. For example, the class dependency 20→19 (stubbing complexity = 0.713) involved in 30 cycles was removed by NC in a certain stage of its execution, as shown in Fig. 8. Both ICD and SC identified 29 similar classes in these cycles, which were highlighted in gray background. We regarded these 29 classes holistically, and constructed a new class dependency 19→25 (stubbing complexity = 0.019 × 29 = 0.551) to represent all dependencies from class 19 towards these similar classes. After similar-class combination, class dependency 19→25 was involved in seven cycles (the actual number of cycles should be 29 × 7 = 203), which was the greatest number of cycles. While the class dependency 20→19 was involved in only two cycles. Hence, NC-ICD and NC-SC removed class dependency 19→25 rather than 20→19, breaking 173 more cycles with 0.162 less stubbing complexity.

The proposed approach yielded fewer OCplx and Stubs for the reduced program *notepad_spl* because of the removal of an extra class dependency 18→19 from the original program. As we observed in the case of RQ2, CWR significantly outperformed NC and SCplx in terms of stubbing cost minimization by calculating the cycles–weight ratio, which considers the two objectives of reducing the total number of test stubs and minimizing the stubbing cost for each removed dependency. While this calculation is easy, it is rough, and it may omit a better class dependency that breaks the same cycles as the initial choice but reduces the stubbing cost, which is the aim of our cycle-breaking algorithm. Although the performance of the proposed cycle-breaking algorithm is superior to that of the existing methods, we cannot guarantee that its results are the best, and especially for some complex programs containing numerous cycles, it is arduous to search all possible solutions. The number of cycles is minimized by means of similar-class combination based on the two proposed properties, which alleviates this issue to some extent. For example, the reduced program *notepad_spl* contained only 30 cycles in Fig. 7, making the problem easier compared with the original 227 cycles.

Interestingly, for *MyBatis*, both SCplx-ICD and SCplx-SC reduced OCplx and Stubs compared with their original versions. SCplx removed more class dependencies without similar-class combination. For example, four similar classes 182, 186, 192, and 193 are identified by ICD and SC. Class dependencies 51→ $182,51$ → $186,51$ →192, and 51→193 were removed first by SCplx due to relatively lower stubbing complexity (the values are all 0.011). However, despite this, it was unable to break all of the cycles in the program, thus necessitating the removal of more class dependencies with higher stubbing complexity. The latter removed class dependencies can also break the same cycles where the first four class dependencies are involved, which can render the stubbing efforts redundant. Moreover, these latter removed class dependencies cannot be considered first due to their higher stubbing complexities. If we regard these similar classes holistically and combine their class dependencies, the stubbing complexity of the newly generated class dependency is 0.044 (4 × 0.011), and this class dependency will not be selected by SCplx. Therefore, SCplx-ICD and SCplx-SC can avoid the construction of extra test stubs.

Overall, similar-class combination can minimize the number of classes (as well as class dependencies) without degrading the performance of the cycle-breaking techniques in terms of minimizing the stubbing cost for most programs. Similar-class combination based on ICD increases the stubbing cost for *GPL_spl* and *JHotDraw* due to the lack of constraints on the coupling values of the class dependencies, which we addressed by combining SCs. Moreover, similar-class combination can alleviate the drawbacks of the competitor approaches in some cases to devise better CITOs with lower stubbing costs. For the proposed approach, it would be easier to find a satisfactory cycle-breaking solution among the fewer cycles obtained by combining SCs.

4.5. Threats to validity

Although the experimental results demonstrate the effectiveness of the proposed approach, some potential threats to the validity of the proposed approach and experiments remain.

Internal validity. In ANS algorithm, a class dependency that is involved in the greatest number of cycles is set as an initialization. The underlying rationale is to accelerate the processing of the algorithm for complex and large cycles. However, the performance of the algorithm may be affected by the initialization, as in the case of *MyBatis*. Therefore, in our subsequent work, we aim to study its detailed effects on performance and explore a better choice for initialization.

Class dependencies of programs are obtained by Soot based on their binary files. Several dynamic class dependencies are neglected because they are formed only when the programs are executed. Depth-first search is adopted to identify cycles in each strongly connected component, but it is difficult to verify these cycles one by one. Therefore, the dependencies among classes and the number of cycles that each class dependency is involved in may not be adequate. Fortunately, this problem has little effect on the comparison between our approach and its competitors because all of the required information is identical for each method.

External validity. Because the three typical cycle-breaking strategies considered herein are not publicly available, we implemented them according to the algorithms provided by Briand et al. [7], Hashim et al. [11], and Abdurazik et al. [14], but made a few minor changes. To emulate the number of cycles in which each class dependency is involved, the algorithm of Briand et al. [7] calculates the product of in-degree of source class and out-degree of target class, while in our experiments, NC directly counts the number of cycles for each

class dependency. To measure the stubbing cost of the test stubs for each class, the algorithm of Hashim et al. [11] counts the number of use relationships from a certain class to other classes. A smaller number of use relationships represents a more independent class, and correspondingly, the test stubs for such classes are easy to construct. Such measurement of stubbing costs is rough and cannot describe the intentions of different class dependencies. For example, creating a test stub for a class dependency that involves multiple method invocations is much more difficult than that for a class dependency that contains only one method invocation. The algorithm of Abdurazik et al. [14] regards the number of parameters and return value types as two different coupling values, but we combine them into attribute coupling in this paper, as described in Section 2.1. Because our experiments mainly focused on different types of cycle-breaking strategies, both SCplx and CWR adopted stubbing complexity, which is the most popular measure.

Our approach was evaluated by applying it to nine Java programs. However, the generalizability of the experimental results to all programs needs further exploration, especially for programs written in other languages. We intend to apply the proposed approach to more programs in the future.

5. Related work

Integration testing is an important part of software testing. Grechanik et al. [20] proposed a novel approach to reduce the number of synthesized integration tests by analyzing the interaction among software components. Tahvili [21] proposed multiple criteria for the approach for test execution optimization in integration testing. CITO generation is an important aspect of integration testing. The existing CITO generation approaches can be divided into graph-based and search-based approaches [6].

Kung et al. [22,23] first proposed a graph-based method to minimize the number of required test stubs. Based on Kung et al.'s work, Tai and Daniels [1] proposed a two-level strategy that assigns a major test order to each class only when considering strongly connected relationships (such as inheritance and aggregations) and determines a minor test order for the classes at the same major test level. Le Traon et al. [24] extracted frond edges from each strongly connected component and removed all of the incoming edges of the node with the maximal number of incoming or outgoing fronds. Differently, Hewett [25] adopted an incremental strategy by including appropriate class candidates into the test order one by one.

Hanh [15] proposed a graph-based method called Triskell that determines class test orders by considering both stub minimization and testing resource allocation. Malloy [26] identified six types of edges in the ORD constructed for C++ programs and assigned different weights to different types of edges to estimate the stubbing costs for such edges. Abdurazik and Offutt [14] proposed nine types of coupling between classes and calculated the stubbing complexity as the weight for each edge. They removed the edge with the maximal cycles–weight ratio by considering both the number of test stubs and their stubbing costs. Bansal [13] combined the above two approaches, introduced new dependencies that were omitted by Malloy, and adopted the cycles–weight ratio to remove edges.

Briand et al. [10] first proposed the search-based method. Borner et al. [27] adopted the simulated annealing algorithm [28] to generate class test orders and selected error-prone class dependencies as the test focus. Multi-objective optimization algorithms were introduced by Vergilio et al. [16] and Assunção et al. [6], such as the ant colony algorithm [29–31] and nondominated sorting genetic algorithm-II [32,33], in attempts to find Pareto-optimal [34] solutions. Guizzo et al. [35,36] introduced a hyper-heuristic [37,38] to choose evolutionary operators that search for class test orders based on their historical performances. Mariani et al. [39] improved Guizzo et al.'s approach [35] by adopting grammatical evolution [40,41] to automatically generate multi-objective evolutionary algorithms. Czibula et al. [42] used reinforcement learning [43,44] to search for a CITO with a minimal number of test stubs. These multi-objective optimization algorithms aim to achieve a good tradeoff between the number of emulated attributes and methods, which is different from the proposed algorithm that aims to minimize the overall stubbing complexity.

The existing approaches are unable to devise satisfactory class test orders because they do not fully consider these two factors, namely the number of test stubs and the corresponding stubbing complexity, in determining the overall stubbing cost. In contrast, the proposed approach addresses this problem and performs better.

6. Conclusion

The existing CITO generation approaches usually require considerable amounts of time to generate a sub-optimal test order for real applications with a moderately large number of classes. We assume that similar-class combination reduces the problem space for CITO generation. Therefore, for the proposed approach, we devised two novel properties – ICD and SC – to identify similar classes involved in the same cycles or having equal test orders. Two propositions were presented to prove the effectiveness of similar-class combination in reducing the problem space for CITO generation. In addition, a cycle-breaking algorithm was proposed to remove the class dependencies in programs by fully considering the two factors that impact the overall stubbing cost, namely the distinct number of test stubs and the corresponding stubbing complexities. A CITO is determined when no cycles exist in the program.

Experiments were conducted to evaluate the performance of the proposed approach against that of three typical cycle-breaking methods (NC, SCplx, and CWR) by applying them to nine Java programs. The experimental results indicated that the proposed approach outperformed the conventional methods on six of the nine programs (up to 13.09% reduction in OCplx), and on the remaining programs, it achieved stubbing costs comparable to those achieved by the conventional methods. The combination of similar classes minimized the number of cycles and classes for eight of the nine programs, which reduced the problem space for the existing methods without degrading their performance in terms of minimizing the stubbing cost.

In this paper, we only identified similar classes that are involved in the same cycles or have equal test priority. However, negative associations exist in different classes when class A is tested and class B should not be tested simultaneously. To further reduce the problem space, in the future, we will investigate other types of relationships that can describe these negative associations between different classes.

Acknowledgments

This work is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (No. 11208017) and the research funds of City University of Hong Kong (7005028 and 7005217), and the Research Support Fund by Intel (9220097), and funding supports from other industry partners (9678149, 9440227, 9440180, 9220103 and 9229029). This project is also supported by the National Research Foundation, Singapore and National University of Singapore through its National Satellite of Excellence in Trustworthy Software Systems (NSOE-TSS) office under the Trustworthy Software Systems - Core Technologies Grant (TSSCTG) award no. NSOE-TSS2019-05.

References

[1] K.C. Tai, F.J. Daniels, Test order for inter-class integration testing of Object-Oriented software, in: Proceedings of the 21st Annual International Computer Software and Applications Conference (COMPSAC'97), 1997, pp. 602–607.
[2] H. Melton, E. Tempero, An empirical study of cycles among classes in Java, Empir. Softw. Eng. 12 (4) (2007) 389–415.
[3] Mockito, 2016, http://site.mockito.org. Online, Accessed 3 Feb.

[4] Junit5, 2017, https://junit.org/junit5. Online, Accessed Sep.

[5] B. Beizer, Software Testing Techniques, second ed., Van Nostrand Reinhold, New York, 1990.

[6] W.K.G. Assunção, T.E. Colanzi, S.R. Vergilio, A. Pozo, A multi-objective optimization approach for the integration and test order problem, Inform. Sci. 267 (2014) 119–139.

[7] 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.

[8] E.L. Lloyd, M.L. Soffa, C.-C. Wang, On locating minimum feedback vertex sets, J. Comput. System Sci. 37 (1988) 292–311.

[9] T. Orenstein, Z. Kohavi, I. Pomeranz, An optimal algorithm for cycle breaking in directed graphs, J. Electron. Test. 7.1-2 (4) (1995) 71–81.

[10] L.C. Briand, J. Feng, Y. Labiche, Experimenting with genetic algorithms to devise optimal integration test orders, in: T.M. Khoshgoftaar (Ed.), Software Engineering with Computational Intelligence, Tech. Rep. TR SCE-02-03, Carleton University, Springer US, Boston, MA, 2003, pp. 204–234.

[11] 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.

[12] T. Hastie, R. Tibshirani, J. Friedman, The Elements of Statistical Learning, second ed., Springer, 2009.

[13] 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.

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

[15] V.L. Hanh, K. Akif, Y. Le Traon, J.-M. Jézéque, Selecting an efficient OO integration testing strategy: An experimental comparison of actual strategies, in: Proceedings of the 15th European Conference on Object-ORiented Programming, Springer Berlin Heidelberg, Berlin, Heidelberg, 2001, pp. 381–401.

[16] S.R. Vergilio, A. Pozo, J.C.G. Árias, R. da Veiga Cabral, T. Nobre, Multi-objective optimization algorithms applied to the class integration and test order problem, Int. J. Softw. Tools Technol. Transf. 14 (2012) 461–475.

[17] R. Tarjan, Depth-first search and linear graph algorithms, in: Proceedings of the 12th Annual Symposium on Switching and Automata Theory (SWAT 1971), 1971, pp. 114–121.

[18] M. Črepinšek, S.-H. Liu, M. Mernik, Exploration and exploitation in evolutionary algorithms: A survey, ACM Comput. Surv. 45 (3) (2013) 1–33.

[19] J.L. Devore, Probability and Statistics for Engineering and the Sciences, fifth ed., Duxbury Press, 1999.

[20] M. Grechanik, G. Devanla, Generating integration tests automatically using frequent patterns of method execution sequences, in: SEKE, 2019, pp. 209–280.

[21] S. Tahvili, Multi-Criteria Optimization of System Integration Testing, Mälardalen University College, Västerås, Eskilstuna, Sweden, GRIN Verlag, 2019.

[22] D.C. Kung, J. Gao, P. Hsia, J. Lin, Y. Toyoshima, Class firewall, test order, and regression testing of Object-Oriented programs, J. Object-Oriented Program. 8 (1995) 51–65.

[23] D.C. Kung, J. Gao, P. Hsia, Y. Toyoshima, C. Chen, On regression testing of Object-Oriented programs, J. Syst. Softw. 32 (1) (1996) 21–40.

[24] Y. Le Traon, T. Jéron, J. Jézéquel, P. Morel, Efficient Object-Oriented integration and regression testing, IEEE Trans. Reliab. 49 (1) (2000) 12–25.

[25] R. Hewett, P. Kijsanayothin, Automated test order generation for software component integration testing, in: Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering (ASE'09), 2009, pp. 211–220.

[26] B.A. Malloy, P.J. Clarke, E.L. Lloyd, A parameterized cost model to order classes for class-based testing of C++ applications, in: Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE'03), 2003, pp. 353–364.

[27] L. Borner, B. Paech, Integration test order strategies to consider test focus and simulation effort, in: Proceedings of the 1st International Conference on Advances in System Testing and Validation Lifecycle, 2009, pp. 80–85.

[28] R.E. Burkard, F. Rendl, A thermodynamically motivated simulation procedure for combinatorial optimization problems, European J. Oper. Res. 17 (2) (1984) 169–174.

[29] K. Doerner, W.J. Gutjahr, R.F. Hartl, C. Strauss, C. Stummer, Pareto ant colony optimization: A metaheuristic approach to multiobjective portfolio selection, Ann. Oper. Res. 131 (1) (2004) 79–99.

[30] M. Dorigo, K. Socha, An Introduction to Ant Colony Optimization, Vol. 194, Computer Science Technical Report No. TR/IRIDIA/2006-010, Universit de Libre de Bruxelles, CP, 2006.

[31] R. da Veiga Cabral, A. Pozo, S.R. Vergilio, A pareto ant colony algorithm applied to the class integration and test order problem, in: Proceedings of the 22nd IFIP WG 6.1 International Conference on Testing Software and Systems, Springer Berlin Heidelberg, Berlin, Heidelberg, 2010, pp. 16–29.

[32] K. Deb, S. Agrawal, A. Pratap, T. Meyarivan, A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II, in: Proceedings of the 6th International Conference on Parallel Problem Solving from Nature (PPSN'00), Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 849–858.

[33] K. Deb, A. Pratap, S. Agarwal, T. Meyarivan, A fast and elitist multi-objective genetic algorithm: NSGA-II, IEEE Trans. Evol. Comput. 6 (2) (2002) 182–197.

[34] V. Pareto, Manuel D'Économie Politique, Ams Press, Paris, 1927.

[35] G. Guizzo, G.M. Fritsche, S.R. Vergilio, A.T.R. Pozo, A hyper-heuristic for the multi-objective integration and test order problem, in: Proceedings of the 2015 Annual Conference on Genetic and Evolutionary Computation, in: (GECCO '15), ACM, New York, NY, USA, 2015, pp. 1343–1350.

[36] G. Guizzo, M. Bazargani, M. Paixao, J.H. Drake, A hyper-heuristic for multi-objective integration and test ordering in Google Guava, in: Proceedings of the 2017 International Symposium on Search Based Software Engineering (SSBSE 2017), Springer International Publishing, Cham, 2017, pp. 168–174.

[37] M. Harman, E.K. Burke, J. Clark, X. Yao, Dynamic adaptive search based software engineering, in: Proceedings of the 2012 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'12), ACM, New York, NY, USA, 2012, pp. 1–8.

[38] E.K. Burke, M. Hyde, G. Kendall, G. Ochoa, E. Ozcan, R. Qu, A Survey of Hyper-Heuristics, Computer Science Technical Report No. NOTTCS-TR-SUB-0906241418-2747, School of Computer Science and Information Technology, University of Nottingham, 2009.

[39] T. Mariani, G. Guizzo, S.R. Vergilio, A.T.R. Pozo, Grammatical evolution for the multi-objective integration and test order problem, in: Proceedings of the 2016 Genetic and Evolutionary Computation Conference, in: (GECCO '16), ACM, New York, NY, USA, 2016, pp. 1069–1076.

[40] C. Ryan, J.J. Collins, M.O. Neill, Grammatical evolution: Evolving programs for an arbitrary language, in: Proceedings of the 1998 European Conference on Genetic Programming, Springer, 1998, pp. 83–96.

[41] E.K. Burke, M.R. Hyde, G. Kendall, Grammatical evolution of local search heuristics, IEEE Trans. Evol. Comput. 16 (3) (2011) 406–417.

[42] G. Czibula, I.G. Czibula, Z. Marian, An effective approach for determining the class integration test order using reinforcement learning, Appl. Soft Comput. 65 (C) (2018) 517–530.

[43] R.S. Sutton, A.G. Barto, Reinforcement Learning: An Introduction, MIT Press, Cambridge, MA, 2011.

[44] L.J. Lin, Self-improving reactive agents based on reinforcement learning, planning and teaching, Mach. Learn. 8 (3–4) (1992) 293–321.