

CITY UNIVERSITY OF HONG KONG

香港城市大學

Integration Testing for Object-Oriented Software:
Enhancing the Efficiency and Quality of the
Automated Test Order Generation Systems

面向對象軟件的集成測試：

提高自動測試序列生成系統的效率和質量

Submitted to

Department of Computer Science

電腦科學系

in Partial Fulfillment of the Requirements
for the Degree of Doctor of Philosophy

哲學博士學位

by

Miao Zhang

張妙

August 2021

二零二一年八月

Abstract

A common problem in the interclass integration testing of object-oriented software is to determine the order in which classes are integrated and tested, which is generally called the class integration test order (CITO) generation problem. The CITO can affect the design and execution of test cases, the order in which integration faults are revealed, and the required test stubs. Therefore, an optimal CITO for an object-oriented program is desired because it can facilitate the detection of integration faults or bugs, and meanwhile save test efforts. However, searching for the optimal CITO is a NP hard problem. Therefore, the existing studies aim to devise a CITO with the lowest possible test efforts.

Despite the encouraging results obtained by the existing approaches, some problems limit the performance and generality of these approaches. One limitation is that these approaches are ineffective and inefficient when generating the CITO for programs with a moderately large number of classes (Problems 1 and 2). Another significant threat is that a test oracle problem exists in these CITO generation systems (Problem 3). In this thesis, we deal with these three problems.

For Problem 1, the unsatisfactory effectiveness is due to the ignorance of the factors that affect the overall test efforts. To address this problem, a new cycle-breaking algorithm ANS is introduced to perform a good Analysis of the relationship between the Number of removed dependencies and the related Stubbing complexity, which are the two main factors

that affect the overall test efforts. 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.

For Problem 2, the inefficiency of the existing approaches for CITO generation is caused by the huge search space, and therefore, we propose a method called CITOSC (Class Integration Test Order generation with Similar-class Combination), which 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.

To address Problem 3, we apply Metamorphic Testing (MT) to validate CITO generation systems and implement it as CITOMT (validating Class Integration Test Order generation systems with Metamorphic Testing). MT has been applied to alleviate test oracle problem in other areas of software testing. Metamorphic Relations (MRs), which are the key components of MT, reason about relations between test outputs of a system. We propose five kinds of MRs to ensure the quality of the CITO generation systems. In these proposed MRs, follow-up test inputs are generated by modifying classes or class dependencies in the source test inputs while some characteristics of the source test outputs are preserved, for example, the same CITO or the equal stubbing cost. Faults can be detected in the CITO systems if an individual MR is violated for certain tests.

Experiments are conducted to evaluate the performance of these proposed methods. The cycle-breaking algorithm ANS achieved reduction of more than 13% in the test efforts, thus outperforming other competing techniques. With similar-class combination, CITOSC minimized the search space. Specifically, it reduced the numbers of classes and class dependencies by over 10% and 6%, respectively, for six of nine used Java programs.

Moreover, for four programs, CITOSC reduced the number of cycles among class dependencies by more than 20%. For CITOMT, failure detection of MT has been successfully demonstrated on three systems implementing the abovementioned competing techniques. On mutation analysis, more than 84% of faulty programs can be detected by all MRs for these CITO generation systems under investigation. To the best of our knowledge, we are the first to introduce MT to validating CITO generation systems. We hope that the promising results will interest and encourage more in-depth research in this field.

Acknowledgements

First of all, I would like to express my sincere gratitude to my supervisor, Dr. KEUNG Wai Jacky, for his great effort and continuous support during my Ph.D. pursuit. His guidance and invaluable advice helped me in all the time of research and writing of this thesis.

I would also like to thank my Ph.D. qualifying panel members, Dr. CHAN Wing Kwong and Dr. YU Yuen Tak, and Prof. CHEN Tsong Yueh, not only for their precious time and insightful comments on my research reports, but also for the hard questions which inspired me to widen my research from various perspectives.

I am very grateful to my dissertation examiners, for their invaluable time spent on carefully reading my thesis, and for providing me with precious suggestions and constructive comments to help improve the quality of this thesis.

My sincere thanks also goes to my friends and lab-mates at City University of Hong Kong, Ms. Yan Xiao, Mr. Shuo Feng, Mr. Md Alamgir Kabir, Mr. Xiao Yu, Mr. Zhen Yang, Mr. Kwabena Ebo Bennin, Ms. Xiaoxue Ma, Mr. Hao Zhang and Mr. Xiupei Mei, for the sleepless nights we were working together before deadlines, and for all the fun we have had in the last three years.

Last but not the least, I would like to thank my family for supporting me spiritually throughout writing this thesis. Without their precious understanding and encouragement, it would not be possible for me to complete my dissertation.

List of Publications

(Listed in reverse chronological order)

Published and Accepted Journal Articles

1. **Miao Zhang**, Jacky Keung, Tsong Yueh Chen, Yan Xiao: Validating class integration test order generation systems with metamorphic testing. **Information & Software Technology** 132: 106507 (2021)
2. **Miao Zhang**, Jacky Keung, Yan Xiao, Md Alamgir Kabir: Evaluating the effects of similar-class combination on class integration test order generation. **Information & Software Technology** 129: 106438 (2021)
3. Shuo Feng, Jacky Keung, Xiao Yu, Yan Xiao, Kwabena Ebo Bennin, Md Alamgir Kabir, **Miao Zhang**: COSTE: Complexity-based OverSampling TEchnique to alleviate the class imbalance problem in software defect prediction. **Information & Software Technology** 129: 106432 (2021)
4. Shujuan Jiang, **Miao Zhang**, Yanmei Zhang, Rongcun Wang, Qiao Yu, Jacky Keung: An integration test order strategy to consider control coupling. **IEEE Transactions on Software Engineering**. (2019)

Published and Accepted Conference Papers

1. **Miao Zhang**, Jacky Keung, Yan Xiao, Md Alamgir Kabir, Shuo Feng: A heuristic approach to break cycles for the class integration test order generation. **COMPSAC 2019**: 47-52, July 2019, Milwaukee, USA
2. Zhen Yang, Jacky Keung, Xiao Yu, Xiaodong Gu, Zhengyuan Wei, Xiaoxue Ma, **Miao Zhang**: A multi-modal transformer-based code summarization approach for smart contracts. Accepted by **ICPC 2021**, May 2021, Madrid, Spain
3. Shujuan Jiang, **Miao Zhang**, Yanmei Zhang, Rongcun Wang, Qiao Yu, Jacky Keung: An integration test order strategy to consider control coupling. Accepted by **ICSE 2020 J1**, July 2020, Seoul, South Korea
4. Zhen Yang, Jacky Keung, **Miao Zhang**, Yan Xiao, Yangyang Huang, Tik Hui: Smart contracts vulnerability auditing with multi-semantics. **COMPSAC 2020**: 892-901, July 2020, Madrid, Spain
5. Md Alamgir Kabir, Jacky Keung, Kwabena Ebo Bennin, **Miao Zhang**: A drift propensity detection technique to improve the performance for cross-version software defect prediction. **COMPSAC 2020**: 882-891, July 2020, Madrid, Spain
6. Md Alamgir Kabir, Jacky Keung, Kwabena Ebo Bennin, **Miao Zhang**: Assessing the Significant Impact of Concept Drift in Software Defect Prediction. **COMPSAC 2019**: 53-58, July 2019, Milwaukee, USA

Contents

Abstract	i
Acknowledgements	iv
List of Publications	v
1 Introduction	1
1.1 Background	2
1.2 Motivation	4
1.2.1 Problems 1&2. Class Integration Test Order Generation Method . .	4
1.2.2 Problem 3. Validation of Class Integration Test Order Generation Systems	6
1.3 Contributions of the Thesis	7
1.4 Organization of the Thesis	9
2 Literature Review	12
2.1 Class Integration Test Order Generation	12
2.1.1 Number of Test Stubs	13
2.1.2 Stubbing Complexity	14
2.1.3 Cycles-Weight Ratio	15

2.2	Metamorphic Testing	16
2.2.1	Basic Knowledge	16
2.2.2	Application of Metamorphic Testing	17
3	ANS: A New Cycle-Breaking Algorithm for the Class Integration Test Order	
	Generation	22
3.1	Introduction	23
3.2	Background	26
3.2.1	Problem Representation	26
3.2.2	Test Stub	27
3.2.3	Stubbing Complexity	28
3.3	Motivation	29
3.4	Algorithm	33
3.4.1	Overview of CITO Generation	33
3.4.2	Overlapping Paths	34
3.4.3	ANS Algorithm	35
3.5	Experimental Design	40
3.6	Experimental Results	41
3.7	Threats to Validity	45
3.8	Conclusion	46
4	CITOSC: Evaluating the Effects of Similar-Class Combination on Class Integration Test Order Generation	47
4.1	Introduction	49
4.2	Motivation	51
4.3	Proposed Approach	52

4.3.1	Overview of the Approach	52
4.3.2	Properties for Similar-Class Identification	54
4.3.3	Similar-Class Combination	57
4.4	Experimental Design	61
4.4.1	Research Questions	61
4.4.2	Experimental Settings and Evaluation Metrics	63
4.5	Experimental Results	63
4.5.1	Research Question 1	63
4.5.2	Research Question 2	67
4.6	Discussion	74
4.6.1	Analysis for Special Cases	74
4.6.2	Threats to Validity	78
4.7	Conclusion	79

5 CITOMT: Validating Class Integration Test Order Generation Systems with Metamorphic Testing

81

5.1	Introduction	82
5.2	Validation Framework	85
5.2.1	Validation Procedure	85
5.2.2	Metamorphic Relations (MRs)	86
5.3	Experimental Design	95
5.3.1	CITO Generation Systems	96
5.3.2	Research Questions	98
5.3.3	Test Case Generation	99
5.3.4	Mutant Generation	101
5.4	Experimental Results	104

5.5	Discussion	113
5.5.1	Experimental Results Analysis	113
5.5.2	Threats to Validity	118
5.6	Conclusion	119
6	Conclusions and Future Work	120
6.1	Summary	120
6.2	Future Work	122
	Bibliography	125

List of Tables

3.1	Comparison of Results for Cycle-Breaking Strategies	31
3.2	Information of Used Programs in Group 1	40
3.3	Information of Used Programs in Group 2	41
3.4	Comparison of Results on Benchmark Programs for ANS and Briand et al.'s methods	42
3.5	Comparison of Results on Programs for ANS and Three Competitors (NC, SCplx and CWR)	44
4.1	Number of instances for similar classes identified by two properties (ICD and SC)	64
4.2	Comparison of Programs Obtained by Combining Similar Classes based on Two Properties	66
4.3	Comparison of Results on Programs After Combining Similar Classes by Property ICD	69
4.4	Comparison of Results on Programs After Combining Similar Classes by Property SC	70
4.5	Class Dependencies Causing Better Result of CWR-ICD for Program log4j3	71
4.6	P-Values for Methods Performing on Programs Before and After Combin- ing Similar Classes	73

4.7	Programs for Which Generated CITO's Are Affected by Similar Class Com- bination	75
5.1	Information of Source Programs	99
5.2	Number of Follow-Up Programs for MRs	100
5.3	Information of Mutants for CITO Generation Systems	102
5.4	Information of Mutants for Cycle-Breaking Module in CITO Generation Systems	103
5.5	Mutation Scores of CITO Generation Systems for Different Test Suites Generated Based on Five MRs	105
5.6	Mutation Scores of Cycle-Breaking Modules for Different Test Suites Gen- erated Based on Five MRs	109
5.7	Mutation Scores of CITO Generation Systems for Different Types of Mutants	110
5.8	Number of Cycles for Each Class Dependency in Source Program and Follow-Up Programs for ATM	114
5.9	Generated CITO's and OCplx by Three Systems for Four Test Suites	115
5.10	Number of Killed Mutants by Different Test Suites	116

List of Figures

1.1	Workflow of CITO	8
3.1	Source Code Example to Describe Coupling Information	27
3.2	A Sample Program to Illustrate Our Motivation	29
3.3	Example of Cycle Breaking	36
4.1	Example for Similar Class Combination	51
4.2	Overview of the Proposed Approach	53
4.3	Relationship between Properties and Similar Classes	65
4.4	Comparison of Object Relation Diagram for notepad__spl: (a) Original Program (b) Reduced Program	67
4.5	Cycles Containing Class Dependency 20→19	76
5.1	Example of MR1 Permutation of Class Names	87
5.2	Example of MR2 Adding/deleting class dependencies for the initial class . .	88
5.3	Example of MR3 Adding/deleting class dependencies for the last class . . .	91
5.4	Example of MR4 Adding a new class with the aim of preserving the origi- nal test order	93
5.5	Example of MR5 Combining classes while preserving the original test order	95
5.6	Overview of Class Integration Test Order Generation Systems	97

5.7	A Snippet to Store Attribute Coupling Values for Class Dependencies in CITO Generation Systems	107
5.8	The Distribution of Killed Mutants for NCSystem	111
5.9	The Distribution of Killed Mutants for SCSystem	111
5.10	The Distribution of Killed Mutants for CWRSystem	112
5.11	Example for New MR (Division) Performed on NCSystem. Class C_4 is divided into two classes (C_4 and C_4').	117

Chapter 1

Introduction

A common problem in the interclass integration testing of object-oriented software is to determine the order in which the classes are integrated and tested [67], which is generally called the class integration test order generation problem. The class integration test order affects the preparation of test cases, and it also determines the order in which inter-class faults are detected [117]. Moreover, different class integration test orders often require different test stubs. A test stub is an incomplete class that provides necessary services for compilation and integration but does not implement the full functionality. Constructing test stubs is an expensive and error-prone operation. Different test stubs require various efforts. However, test stubs cannot be avoided because circular dependencies are quite common [86], especially in large-scale programs. Therefore, an optimal class integration test order for an object-oriented program is desired because it can save redundant efforts in constructing test stubs for integration testing.

Research on class integration test order generation is of great importance in real life. As the crowdsourcing in software development becomes gradually popular recently, many program components, such as classes in object-oriented programs, are divided into different

participants to implement, and finally these classes are combined to achieve a cumulative result. How to combine these classes quickly and accurately is a question, and class integration test order generation can provide a good solution for this issue. However, the existing publications on class integration test order generation still have some limitations.

This thesis is devoted to the study of class integration test order generation. In this chapter, we first briefly introduce the background of class integration test order generation research. Then we provide the motivation and contribution of our work.

1.1 Background

In software testing for object-oriented programs, integration testing is an important process to verify whether all classes could cooperate well when they are assembled into the whole program. Determining an integration test order for all classes is a common problem encountered during integration testing. Such test orders can affect the design and execution of test cases, the order in which integration faults are revealed, and the required test stubs. Therefore, an optimal class integration test order can facilitate the detection of integration faults or bugs, and meanwhile save test efforts, which means not only less complex test stubs, but also fewer test cases.

The ideal solution for the class integration test order generation problem is an optimal test order with a minimal stubbing cost, which is important but also difficult. Therefore, the existing studies aim to devise a class integration test order with the lowest possible test efforts. These studies could be characterized as graph-based and search-based approaches.

For graph-based methods, a diagram (for example, Object Relation Diagram or Test Dependency Graph) is constructed for the program where each vertex represents a class and each edge represents the dependency between classes. Such methods try to find a best

way to traverse all vertices in the diagram. It is obvious that all vertices can be visited by reverse topological sorting when no circular dependencies exist in the program, and consequently, a class integration test order can be generated according to the visiting order of each vertex. However, cycles are common in the program due to complex dependencies between classes. In order to transfer a cyclic diagram to an acyclic diagram, some edge removing rules are designed to break cycles, such as removing the edges that are involved in more cycles or removing the edges with the minimal stubbing cost.

Search-based methods adopt evolutionary algorithms to generate class integration test orders. In such methods, a class integration test order is represented by an individual, and plenty of individuals compose an initial population. Evolutionary operations, such as mutation and crossover, are adopted to parental individuals to generate offspring, i.e., more class integration test orders as alternatives. A Fitness function is proposed to choose the best solution (i.e., the class integration test order with the minimum testing efforts) from the search space. The above process will be repeated until the number of iterations is exceeded.

Although these existing methods have made considerable progress, there are some limitations that prevent them from achieving better performance. Moreover, the systems implemented based on these methods are untestable. There is no relevant research trying to validate such systems.

This thesis aims at enhancing the cost-effective for class integration test order generation. The first phase of studies is the generation of class integration test orders with the minimal stubbing cost, followed by a second phase that aims to verify the systems for class integration test order generation.

1.2 Motivation

Despite the encouraging results obtained by the existing approaches, some problems limit the performance and generality of these approaches. One limitation is that these approaches are inefficient and ineffective for programs with a moderately large number of classes. Another significant threat is that a test oracle problem exists in the class integration test order generation systems. We describe these three problems in detail below.

1.2.1 Problems 1&2. Class Integration Test Order Generation Method

Researchers have proposed different kinds of approaches for the class integration test order generation. Although these approaches have achieved satisfactory results for some small or medium programs, such approaches are ineffective and inefficient when dealing with the programs that contain a sheer volume of cycles or classes. As we have introduced before, the existing approaches for the class integration test order generation can be characterized into two types: search-based and graph-based. We will explain two problems for these two kinds of approaches respectively.

- **Problem 1. Ineffective**

- Searching for the optimal class integration test order is a NP hard problem. Therefore, the existing approaches try to devise the sub-optimal class integration test order instead. However, sometimes the obtained solutions by the existing approaches are unsatisfactory due to the ignorance of the factors that affect the overall stubbing cost. The overall stubbing cost measures the total stubbing efforts during the whole integration testing, which can be affected by two factors. One is the number of test stubs required, and the other is the stubbing

cost for each test stub. However, the existing approaches ignore these two factors when they are generating class integration test orders, which makes these methods ineffective.

- Specifically, for graph-based methods, edges are removed to break cycles that are formed by the class dependencies. Usually, the edge with the greatest number of cycles will be removed, which only considers the first factor (the number of test stubs), or the edge with the minimal stubbing cost is deleted, which only considers the second factor. Similarly, for search-based methods, a fitness function is used to calculate the overall stubbing cost for a complete class integration test order, which cannot perform a good analysis between two factors.

- **Problem 2. Inefficient**

- The existing approaches are able to generate class integration test orders for those small or medium programs, but they usually take a large amount of time to generate their sub-optimal solutions for real-life applications with such a moderately large number of classes. As the number of classes increase, the dependencies between classes will become complex, and correspondingly, the number of cycles will also increase. For graph-based methods, the critical step to generate class integration test order is to break all cycles. It is obvious that it would be more difficult to break hundreds or thousands of complicated cycles compared with those simple ones. For search-based methods, as the search space is composed by all the possible solutions (i.e., class integration test orders), the search space is well-known to be $N!$, where N denotes the number of classes in the program under integration testing. Therefore, the search space will become great as the number of classes increases.

To address these two limitations, a new class integration test order generation approach is required. For problem 1, to improve the effectiveness we aim to propose a new cycle breaking algorithm, which try to perform a good analysis between two factors that affect the overall stubbing cost, i.e., the number of test stubs and the stubbing cost for each test stub.

For problem 2, the problem space for the large-scale programs can be minimized by reducing the number of classes. We develop a new notion of similar classes. Based on this new concept, we propose two properties to identify such similar classes. Similar classes can be combined, and the corresponding class dependencies can be effectively identified and some redundant cycles among classes can be removed to enhance the efficiency of the class integration test order generation.

1.2.2 Problem 3. Validation of Class Integration Test Order Generation Systems

Although many researchers have proposed different kinds of approaches to generate class integration test orders, no studies try to verify the correctness of these systems implemented based on those proposed approaches. This is because a test oracle problem exists in the class integration test order generation systems. Devising an optimal class integration test order is a NP hard problem, therefore, many existing approaches try to search for a sub-optimal solution instead, which is a local optimal solution. As we cannot define what is a local optimal, we are unable to determine whether the generated class integration test order is the actual output that should be devised by the class integration test order generation system. In other words, such systems have the test oracle problem. It becomes difficult to test these class integration test order generation systems without the test oracle. Fortunately, metamorphic testing has been applied to alleviate test oracle problem in software testing,

and they have been adopted in other fields in software engineering, such as, validation, quality evaluation, as well as program analysis and comprehensive. Metamorphic relations are the key components in metamorphic testing, which are used to verify the relations among the inputs and outputs of multiple executions of the system under test. To fill the gap that lacks the validation of systems implemented based on these proposed class integration test order generation approaches, we adopt metamorphic testing on class integration test order generation systems. By adopting metamorphic relations, we can better analyse the class integration test orders generated by these existing systems, identify potential bugs or errors in such systems, and subsequently fix these problems and generate better class integration test orders.

1.3 Contributions of the Thesis

The proposed framework in this thesis aims at enhancing the cost-effectiveness of integration testing for object-oriented programs, through the improvement of efficiency and effectiveness of the systems that generate class integration test orders, and the use of metamorphic relations to verify the existing typical systems for class integration test order generation. The entire workflow is illustrated in Figure 1.1.

The main contributions of this thesis are as follows:

- A cycle-breaking algorithm is proposed to comprehensively consider the two factors that affect the overall stubbing cost, namely the number of test stubs and the corresponding stubbing cost for each test stub. This algorithm aims to keep a good trade-off between these two factors, which is usually ignored by the existing approaches for class integration test order generation.

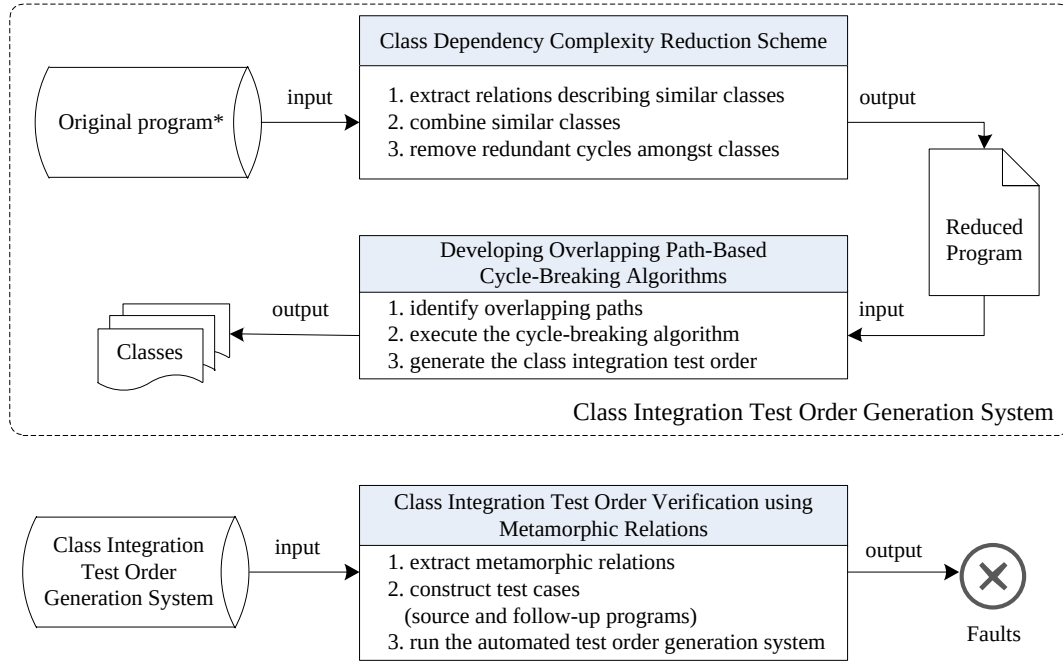


Figure 1.1: Workflow of CITO

- Based on the hypothesis that similar-class combination can reduce the problem space for large-scale programs that contain a moderately large number of classes, we apply the idea of similar-class combination to class integration test order generation, which can reduce the number of classes as well as the number of cycles among classes.
- A new notion of similar classes has been developed through the identification of some characteristics about class dependence, namely, identical class dependence and symmetric classes. Two propositions are elucidated to verify the effectiveness of similar-class combination.
- Experimental results confirmed the effects of similar-class combination on the minimization of the problem space for the class integration test order generation. The

proposed cycle-breaking algorithm in this thesis outperforms other competing methods in minimizing the stubbing costs of the generated class integration test orders.

- CITOMT, a verification framework for class integration test order generation systems is proposed. To our best knowledge, this is the first time to validate such kind of systems, and it is also the first time to introduce metamorphic testing in integration testing.
- Five kinds of metamorphic relations are proposed to test the class integration test order generation systems effectively, which can be extended into all systems to generate class integration test orders. Before that, no references to the applicable metamorphic relations can be obtained for testing the class integration test order generation systems.
- Experiments are conducted to demonstrate the failure detection capability of metamorphic testing. The experimental results show that the proposed metamorphic relations are able to systematically and effectively detect faults in the class integration test order generation systems. This study explores a new application domain in metamorphic testing and further extends its applications in software testing.

1.4 Organization of the Thesis

The rest of this thesis is organized as follows.

In Chapter 2, we first review related work on the class integration test order generation, including some basic knowledge and the previously reported class integration test order generation approaches. Then we briefly describe metamorphic testing and its applications in the context of software engineering.

To improve the effectiveness of the existing approaches, a new cycle-breaking algorithm called ANS is introduced in Chapter 3 to minimize the stubbing cost of the generated test orders. ANS performs a good Analysis of the relationship between the Number of removed dependencies and the related Stubbing complexity. Therefore, these two factors that impact the stubbing cost are fully considered in ANS, which is the main novelty in this algorithm. Experimental results show that the proposed cycle-breaking algorithm outperforms other competing methods.

To improve the efficiency of the existing approaches, we introduce a new method in Chapter 4 and implement it as CITOSC, a class integration test order generation approach with similar-class combination. We hypothesis that similar-class combination can remove class dependencies and yield a smaller problem space for class integration test order generation. Two properties are proposed to identify similar classes. To scientifically test this hypothesis and evaluate the performance of the proposed algorithm, a field experiment is conducted to compare CITOSC with its counterparts. The empirical results validate the positive effects of similar-class combination on minimizing the number of classes as well as class dependencies in programs under test.

Chapter 5 describes CITOMT in detail. We introduce a validation framework for class integration test order generation systems using metamorphic testing, which is the first time to validate such integration testing systems. It is also the first time that metamorphic testing is applied in integration testing. In our framework, we propose five MRs to validate such systems. We evaluate the capability of our framework in detecting faults for class integration test order generation systems using mutation analysis. The experimental results show that more than 84% of faulty programs can be detected by all metamorphic relations for all systems under test.

We conclude the thesis in Chapter 6, which includes the results that have been drawn from the research, main contributions, and suggestions for future work.

Chapter 2

Literature Review

Class integration test order (CITO) generation is a common problem in the interclass integration testing. Different test orders often lead to different test stubs and creating a test stub is an expensive and error-prone operation. Therefore, an optimal class test order is desired because it can save redundant efforts in constructing test stubs. Several publications have proposed various approaches and strategies for the CITO generation problems.

In this chapter, we first describe related work on CITO generation. Although many approaches are proposed to address CITO generation problems, there are little techniques or methods applied to validate the corresponding CITO generation systems. Therefore, we discuss metamorphic testing in the context of software engineering, which is used to validate the CITO generation systems in this thesis.

2.1 Class Integration Test Order Generation

Integration testing is an important part of software testing. Grechanik et al. [47] proposed a novel approach to reduce the number of synthesized integration tests by analyzing the interaction among software components. Tahvili [116] proposed multiple criteria for the

approach for test execution optimization in integration testing. CITO generation is an important aspect of integration testing.

Works that address CITO problems consist of graph-based and search-based methods, which can also be characterized by their measurement of stubbing cost: the number of stubs, the stubbing complexity, and the cycles-weight ratio.

2.1.1 Number of Test Stubs

Kung et al. [67, 68] first used the number of test stubs to assess CITO and were the first to propose a graph-based method for the CITO generation problem. They constructed an Object Relation Diagram (ORD) and identified all cycles in the program. CITO can be generated by topological sorting if no cycles exist in the diagram; otherwise, some edges must be deleted to break the cycles before that.

Based on Kung et al.'s work, Tai and Daniels [117] 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. [72] 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 [57] adopted an incremental strategy by including appropriate class candidates into the test order one by one.

Briand et al. [12] improved the approach used by Kung et al. and assigned weight for each edge to roughly estimate the number of cycles that it is involved. The edge with the highest weight is removed because it breaks the most cycles.

Hanh [52] proposed a graph-based method called Triskell that determines class test orders by considering both stub minimization and testing resource allocation.

2.1.2 Stubbing Complexity

Malloy [81] 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.

Briand et al. [11] first proposed the measurement method called stubbing complexity for the overall stubbing cost of the generated CITO. They also first proposed the search-based method. After that, various kinds of evolutionary algorithms, such as the simulated annealing algorithm [13, 113] and particle swarm optimization, were applied, and some performed better performance than the traditional ones.

Borner et al. [10] adopted the simulated annealing algorithm [13] 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. [123] and Assunção et al. [5], such as the ant colony algorithm [41, 43, 35] and non-dominated sorting genetic algorithm-II [36, 37], in attempts to find Pareto-optimal [95] solutions. Guizzo et al. [50, 49] introduced a hyper-heuristic [53, 15] to choose evolutionary operators that search for class test orders based on their historical performances. Mariani et al. [83] improved Guizzo et al.'s approach [50] by adopting grammatical evolution [104, 14] to automatically generate multi-objective evolutionary algorithms. Czibula et al. [34] used reinforcement learning [115, 74] to search for a CITO with a minimal number of test stubs. These multi-objective optimization algorithms aim to achieve a good trade-off between the number of emulated attributes and methods. They used the number of emulated attributes and methods to assess the stubbing cost, which is different from the proposed algorithm that aims to minimize the overall stubbing complexity.

2.1.3 Cycles-Weight Ratio

Abdurazik and Offutt [3] 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.

Hashim [55] estimated interclass coupling strengths by the numbers of attributes and methods involved in interclass interaction. The fewer attributes or methods involved indicates a weak coupling strength between classes and thus a lower stubbing cost.

Bansal [6] combined the two measurements of stubbing cost mentioned above (i.e., the stubbing complexity and the number of stubs). He also introduced new dependencies that were omitted by Malloy. For each edge, he estimated the stubbing complexity and the number of cycles in which it is involved, deleting the edge with minimum stubbing complexity and involved in more cycles.

Wang et al. [124] proposed a multigranular flow network model to describe inter-class relationships, and identified critical classes that are frequently executed, have more complexity, and are appreciably influential in the system. They evaluated each edge from three new aspects other than the stubbing complexity: vulnerabilities, threats, and failure consequences. The edge with the greatest weight was removed, and critical classes would be integrated first.

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. Therefore, we aim to propose new approach to address this problem and perform better.

2.2 Metamorphic Testing

The test oracle problem and the reliable test set problem are two fundamental problems that need to be addressed in software testing. The test oracle problem [7, 92, 96, 98] refers to the situations where the execution result is difficult to be verified for the program given a test case (that is, an input selected to test the program). For the reliable test set problem, it is usually impossible to execute all the existing test cases, therefore, it is important to effectively select a subset of test cases, that is, the reliable test set, which can determine whether the program is correct or not. Some testing techniques have been proposed to generate test cases for solving the reliable test set problem, such as random testing [51], coverage testing [139], search-based testing [54], and symbolic execution [16]. Also, assertion checking [103] and N-version programming [82] are proposed to address the test oracle problem. Unlike the aforementioned testing techniques that can only address one of two problems, metamorphic testing (MT) is able to address both fundamental problems. Moreover, researchers conducted experiments to compare the performance of MT with that of assertion checking [58, 132] for addressing the test oracle problem. The results show that MT is able to detect more faults than assertion checking although the computational cost is a slightly higher.

We first introduce the basic knowledge on MT, and then describe its application in different domains, especially in software engineering and testing.

2.2.1 Basic Knowledge

Metamorphic Testing is first proposed by Chen et al. [26, 21] to alleviate the problem of test oracles by verifying the relations among the inputs and outputs of multiple executions for the programs or systems under testing. Metamorphic Relation (MR) is the key component

in MT, which reasons about relations between test outputs of a program when it lacks test oracles. In general, a MR is a necessary property of a function f over function inputs $x_1, x_2, \dots, x_n$ ($n > 1$), and their corresponding output values $f(x_1), f(x_2), \dots, f(x_n)$ [30, 107]. For example, a mathematical property of the *sine* function is that $\sin(x) = \sin(\pi - x)$, the corresponding MR is that if two function inputs x_1 and x_2 satisfy $x_1 + x_2 = \pi$, then two function outputs should be equal, i.e., $\sin(x_1) = \sin(x_2)$. In this example, x_2 is constructed based on $\langle x_1, \sin(x_1) \rangle$ according to the MR, therefore, we refer to x_1 as a source input and x_2 as a follow-up input. Such relations can be used to test the program when we are not sure whether the test output is correct. An implementation for this function must have faults if it violates the above relation.

The biggest difference is that conventional testing approaches only focus on each individual test output, while MT verifies MRs among the inputs and outputs of the multiple executions where MRs specify how the change to the test input affects the test output. Such MRs have been proved by a general framework proposed by Chen et al. [32]. MT detects faults in the programs that violate MRs, therefore, it can alleviate the test oracle problem. Researchers have conducted surveys on MT and presented an overview of the application of MT in different domains [42, 59, 107].

2.2.2 Application of Metamorphic Testing

Many studies from different domains have proved the effectiveness of MT to alleviate the test oracle problem. An empirical study [76] shows that a small number of the proposed MRs can achieve very encouraging results in the detection of the software faults, covering various domains, such as, bioinformatics [27, 100, 101, 105], embedded systems [22, 61, 69], web services [114, 23, 19, 20], simulation and modelling [40, 91, 17], databases [75, 39], components [9, 80], compilers [71, 118], online search functions and

search engines [135, 137, 136], machine-learning classifiers [88, 90, 126], software product lines [108, 109], computer graphics [85, 48, 24, 25, 65] and security-critical applications (cybersecurity [29] and cryptography [99, 87]).

Moreover, MT has helped to detect many real-life faults in the abovementioned domains. In computer graphics, the binary space partitioning tree was used to solve the surface visibility problem. Kuo et al. [70] adopted MT to verify a binary space partitioning tree program and detected a real bug in it. In embedded systems, Jiang [61] proposed several MRs to test the correctness of a Central Processing Unit (CPU) scheduling program and detected two real bugs in an open source simulator [120]. Olsen and Raunak [93] validated the simulation models through MT. Segura et al. [106] evaluated the effectiveness of MT's fault detection capability in variability analysis tools and detected 19 real bugs in seven out of the 15 tools under test. In machine learning, Murphy et al. [89] put forward six kinds of MRs (that is, addition, permutation, multiplication, inclusion, exclusion, and inversion) for the verification of machine learning application, and discovered real bugs in three specific machine learning tools. Xie et al. [125, 85] presented MRs to detect faults in two supervised classifiers, i.e., k-Nearest Neighbor (KNN) and Naïve Bayes. The experimental results revealed some real faults in the open source machine learning tool Weka [45].

MT has become more popular due to more functions recently, such as, data generation and user expectations inspection, which extends beyond fault detection. In image classification, MRs were used for data augmentation to enhance the performance of classifiers [130] and identify implementation bugs [44]. In autonomous driving systems [133], MRs were applied to predict the driving behaviours, i.e., the driving behaviours should be identical whatever weather conditions are on the same road [131], Zhou et al. [134] extracted valuable MRs to help users to better understand and use modern information

technology paradigms, such as commercial websites, google maps navigation, and video analysis software, which discovered some unreasonable settings that conflict with user expectations. In addition to the abovementioned domains, MT has been more widely applied in software engineering [4, 32, 33, 62, 64, 78, 102, 129, 128], especially in software testing [9, 22, 23, 27, 29, 73, 71, 75, 109, 126, 137]. Chen et al. [33] integrated MT with symbolic evaluation for program proving. In their method, symbolic execution was adopted to identify whether the inputs for the program violated the proposed MRs. The necessary conditions of program correctness can also be proved by this method.

Liu et al. [78] applied MT into fault tolerance. Fault tolerance aims to enable the program performs its functions successfully even though some faults still exist. A trustworthy input can be determined by the number of satisfied and violated MRs, that is, an input is trustable if it satisfied more MRs and violated fewer MRs. Jin et al. [64] combined both concolic testing [46, 112] and MT with branch-switching debugging. A synthesized program was generated based on the original program using MR check. Then concolic testing and branch-switching debugging were adopted to localize the potential faults. The experimental results demonstrated the fault localization ability of the proposed approach. Lidbury et al. [73] applied equivalence modulo inputs (EMI testing) to detect faults in the OpenCL compilers. EMI testing [71] generated variants of the original program based on MRs, so that the outputs of such variants could be predicted. Compilation errors can be discovered based on the different behaviors that should be identical for some given test inputs.

In performance testing, Segura et al. [111] extracted MRs between performance measurements of different executions of the program and applied MT to reveal performance failures. They also adopted MRs to detect faults in RESTful Web APIs [110]. Pesu et al. [97] evaluated the performance of different machine translation services by the MR that an

ideal translator should provide the same translation results whether or not via an intermediate language. Jiang et al. [63] replaced the original test suite by the group of test cases associated with MRs, to guide the process of automated program repair.

Xie et al. [127] combined MT with spectrum-based fault localization technique to estimate the probability of each program statement of being faulty. Different from the conventional testing technique that checks whether the actual output of test cases is correct or not, their approach determines whether the source and follow-up inputs and outputs violate the proposed MRs. The results showed that three real-life faults were detected by MT in three out of seven programs [102, 128] in the Siemens suite [60].

Although the usefulness of MT has been widely recognized, there are still some challenges that had not been addressed and some opportunities for future research about MT that are gradually emerging [30].

Construction of MRs is still a labor-intensive task and requires the help from a domain expert or a programmer. Several studies were conducted to systematically identify MRs [77] and select effective MRs [18, 28, 84]. Chen et al. [31] devised a framework based on category-choice to identify MRs. Zhu [138] implemented a Java unit testing tool based on MT called JFuzz. In their tool, data mutation operators were adopted to devise MRs automatically. Troya et al. [121] investigated how to infer likely MRs for model transformations automatically. Kanewala et al. [66] believed that functions owned the similar control flow and data dependency graphs were most likely to have similar MRs. Therefore, they proposed machine learning based approaches to predict MRs. For a given function, features were extracted using graph kernels from its control flow graphs and data dependency information.

Different MRs own the different capability of fault detection. For example, Le et al. [71] used a very simple MR and detected 147 unique bugs in two popular C compilers

(GCC and LLWM) in 2014. But they are not the only researchers to use MT to test compilers. In 2010, Tao et al. [118] also conducted the testing of similar compilers, but they only found one fault in the GCC compiler. The number of faults detected by two research teams is dramatically different, which means MRs have a significant impact on the MT's capability for fault detection. Furthermore, as observed by Liu et al. [76], a small number of diverse MRs may effectively reveal almost all faults. Therefore, we aim to analyse different kinds of MRs, develop more effective MRs, and demonstrate the practicality and robustness of class integration test order generation systems through a comprehensive scaled experiment.

Chapter 3

ANS: A New Cycle-Breaking Algorithm for the Class Integration Test Order Generation

It is a general objective to minimize overall stubbing cost when performing class integration test order generation. The overall stubbing cost is affected by two factors: the number of required test stubs and the stubbing complexity. However, existing approaches are unable to obtain a satisfactory class integration test order, this is largely due to the lack of a comprehensive trade-off analysis between the aforementioned two factors.

To minimize overall stubbing cost by balancing these two factors, we propose a new cycle-breaking algorithm called **ANS**, which aims to keep a good **Analysis** involving the **Number** of removed dependencies and the related **Stubbing** complexity. Firstly, we select a class dependency as the initialization mechanism to remove cycles. Then we search for a near ideal set of class dependencies to replace the initialized class dependency, such

dependencies break the same or greater number of cycles as the initialized dependency but attract less stubbing cost.

Empirical experiments are conducted on nine open-source Java programs to evaluate the performance of the proposed algorithm. The results show that ANS avoided the limitation in the existing cycle-breaking strategies, and achieved reduction of more than 13% in the stubbing cost, thus outperforming other competing techniques. ANS is capable of generating class test orders with significantly lower stubbing cost.

The rest of this chapter is organized as follows. Section 3.1 introduces the whole work of this chapter. Section 3.2 describes the background. Section 3.3 introduces the motivation. We present our algorithm in Section 3.4. Section 3.5 presents the detailed design of our experiments and Section 3.6 summarizes the results. Section 3.7 discusses the possible threats to validity. Section 3.8 concludes this chapter.

3.1 Introduction

Stubbing is inevitable in the overall testing process, since some methods, classes or subsystems might not be available when we test other components that depend on them. For a class dependency, a test stub is built for the source class, emulating the behaviors of the dependent class. In integration testing, the order in which classes are integrated and tested has a significant impact on the construction of test stubs. For example, when class *A* depends on class *B*, we can integrate class *B* before class *A* without any test stubs. But 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*. As shown in this example, different test orders lead to different test stubs, and corresponding various stubbing cost. If classes are independent of each other,

they can be tested directly, but in real life, class dependency is a common phenomenon. Especially, when cycles involved in large programs that cross different classes, an order should be decided which classes are tested first. Since we cannot avoid stubbing when we conduct testing, minimizing overall stubbing cost is extremely important when generating class integration test orders (CITOs).

Although many tools to easily create test stubs (or mocks) have been implemented, for instance, Mockito [1] and JUnit [2], test stubs are error-prone [8], also, how to design test stubs is still a question for tester. Therefore, we require an optimal test order to achieve the minimal stubbing cost, and based on this test order to implement corresponding test stubs via these tools.

Generating the optimal CITO is a NP-hard problem, but researchers have proposed many techniques to generate CITOs with the lowest possible stubbing cost. However, the performances of the existing CITO generation approaches are unsatisfactory in some cases, which are caused by the following problem:

- Ignorance of the factors that affect the overall stubbing cost makes these methods ineffective.

As suggested by Briand et al. in [12], "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 [11]. Various class test orders cause different overall stubbing cost. For the entire integration testing, the overall stubbing cost is the sum of the stubbing cost for all test stubs, which 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. [12] 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. [55] focused only on the second factor and sequentially created test stubs for the class dependencies with the minimal stubbing costs. Other cycle-breaking strategies considered both two factors, such as, Bansal et al. [6] and Abdurazik et al. [3] used cycles-weight ratio to measure the stubbing cost for each class dependency. The edge with a higher cycles-weight ratio value indicates that it is involved in more cycles but attracts minimal stubbing cost if it is removed. Calculation of cycles-weight ratio can be helpful, but it is too rough and in some cases is unable to keep a good trade-off analysis between the number of created stubs (cycles) and the corresponding stubbing complexity (weight). The same problem is encountered in the search-based approaches due to the lack of sufficient guidance in the search process. More details will be described in Section 3.3.

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. 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.

3.2 Background

In this section, we introduce the basic knowledge about CITO generation, including the problem representation, test stubs and stubbing complexity.

3.2.1 Problem Representation

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

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

Class Integration Test Order (CITO) Generation Problem: CITO generation problem aims to find a solution $o_s = \{C_{s1}, C_{s2}, \dots, C_{sm}\}$ 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, $Dep = \{dep_1, dep_2, \dots, dep_k\}$ 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 $Dep_s = \{dep_{s1}, dep_{s2}, \dots, dep_{sk}\}$ 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.

3.2.2 Test Stub

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, such as, a variable from the target class that is used as a parameter in the source class's method, or a method from the target class that is invoked by 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. Specifically, the value of attribute coupling between class *A* and *B* is equal to the number of attributes (including variables, different parameters and return types) of class *B* that are accessed by class *A*. Similarly, method coupling between class *A* and *B* is the number of methods of class *B* that are called by class *A*.

1	<code>public class A {</code>	1	<code>public class B {</code>
2	<code> public A() {</code>	2	<code> public B() {</code>
3	<code> B b = new B();</code>	3	<code> int b1 = 1;</code>
4	<code> int a1 = 1;</code>	4	<code> double b2 = 1.0;</code>
5	<code> double a2 = 0.1;</code>	5	<code> }</code>
6	<code> }</code>	6	<code> public int getInt(){</code>
7	<code> public int methodA1() {</code>	7	<code> return b1;</code>
8	<code> return b.getInt();</code>	8	<code> }</code>
9	<code> }</code>	9	<code> public double getDouble() {</code>
10	<code> public double methodA2(B b) {</code>	10	<code> return b2;</code>
11	<code> a2 = a1 + b.getDouble();</code>	11	<code> }</code>
12	<code> return a2;</code>	12	<code>}</code>
13	<code> }</code>		
14	<code>}</code>		
15			

Figure 3.1: Source Code Example to Describe Coupling Information

Figure 3.1 shows a source code example to describe the coupling information between class *A* and class *B*. In Figure 3.1, 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).

3.2.3 Stubbing Complexity

Briand et al. [11] 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 attribute accesses and method invocations, respectively. The stubbing cost is calculated by:

$$SCplx(i, j) = [W_A \cdot \overline{A(i, j)}^2 + W_M \cdot \overline{M(i, j)}^2]^{1/2} \quad (3.1)$$

Because attribute coupling and method coupling may vary significantly within the same program, we normalize them using min-max normalization [56] to avoid the influences of the extreme values. In Equation 3.1, the normalized attribute coupling and method coupling are linear-weighted, and their weights are equal to 0.5. For the program shown in Figure 3.1, given the minimal attributes and methods are zero, the maximal attributes and methods are two and four, respectively, the normalized attribute coupling and method coupling are 0.5 and 0.75, respectively. Corresponding, the stubbing complexity of creating a test stub for class *A* to emulate the behaviors of class *B* is 0.64 according to the above Equation 3.1.

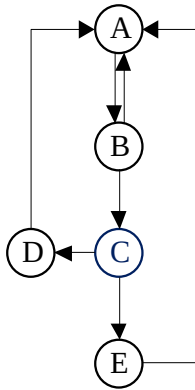
For a generated CITO ϕ , the overall stubbing complexity (OCplx) is used to estimate the overall stubbing cost through integration testing. OCplx is equal to the sum of the stubbing complexity of each test stub that belongs to the set *Stubs*, and it is calculated as follows:

$$OCplx(o) = \sum_{(i,j) \in Stubs} SCplx(i,j) \quad (3.2)$$

According to this equation, 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.

3.3 Motivation

In this section, 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 algorithm to address this problem.



(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

Figure 3.2: A Sample Program to Illustrate Our Motivation

Figure 3.2(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 Figure 3.2(a), this sample program contains three cycles.

- $A \rightarrow B \rightarrow A$
- $A \rightarrow B \rightarrow C \rightarrow D \rightarrow A$
- $A \rightarrow B \rightarrow C \rightarrow E \rightarrow A$

Table (b) in Figure 3.2 summarizes the information pertaining to class dependencies in this sample program. Column 2 represents the value of attribute coupling and method coupling for the related class dependency. Column 3 counts the number of cycles in which the class dependency is involved. Column 4 calculates the stubbing complexity for the class dependency according to Equation 3.1. The last column represents the cycles-weight ratio, which is calculated by dividing the number of cycles by the value of stubbing complexity. 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. [12] 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. [55] 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. [6] and Abdurazik et al. [3]. 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 3.1 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 Equation 3.2. The optimal choices for class dependency removal are also listed.

Table 3.1: 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

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 [11, 52] 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 [5, 123] consider only the number of emulated methods and attributes, but they omit the detailed information. For 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 chapter, 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.

3.4 Algorithm

We first introduce the whole process of graph-based methods for CITO generation, then analyze the factors that impact the performance of the existing cycle-breaking algorithms, and finally introduce our cycle-breaking algorithm, i.e., ANS.

3.4.1 Overview of CITO Generation

First, an ORD is created to represent class dependencies for the program under the test. As shown in Figure 3.2(a), each value in the node represents the serial number of the corresponding class, and each edge represents the corresponding class dependency. Stubbing complexity is calculated for each dependency based on the input attribute coupling and method coupling. Therefore, the CITO generation problem is transformed into the problem of traversing nodes in the ORD.

Then, the Tarjan's algorithm [119] is adopted to execute depth-first search for the ORD and divide the diagram into multiple strongly connected components. A strongly connected component containing more than one node is regarded as non-trivial. Cycles can only exist in such a non-trivial strongly connected component. That is because all pairs of vertices in a cycle are connected, these vertices must be in the same strongly connected component.

So, we regard such strongly connected component as the smallest element to deal with, and identify all cycles in it.

For each non-trivial strongly connected component, the ANS algorithm is applied to generate the dependencies that are going to be removed. The corresponding test stubs are built for these dependencies. Once all cycles in the ORD are broken, nodes in the ORD can be visited by reverse topological sorting, and consequently, the CITO is generated based on the node visitation order.

3.4.2 Overlapping Paths

The cycles among the class dependencies in the program 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 1: 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 $c_A = C_{A1} \rightarrow C_{A2} \rightarrow \dots \rightarrow C_{Am} \rightarrow C_{A1}$, and similarly, c_B consists of n unique classes $c_B = C_{B1} \rightarrow C_{B2} \rightarrow \dots \rightarrow C_{Bn} \rightarrow C_{B1}$. For a path $P_A = C_{Ai} \rightarrow C_{A(i+1)} \rightarrow \dots \rightarrow C_{Aj}$ ($1 \leq i < j \leq m$) in c_A , if there exists a path in c_B , i.e., $P_B = C_{Bi} \rightarrow C_{B(i+1)} \rightarrow \dots \rightarrow C_{Bj}$, 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.

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 3.3.

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.

3.4.3 ANS Algorithm

We propose a new cycle-breaking algorithm called **ANS** because the basic idea of the proposed algorithm is to perform a good **A**nalysis of the relationship between the **N**umber of removed dependencies and the related **S**tubbing 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 [122].

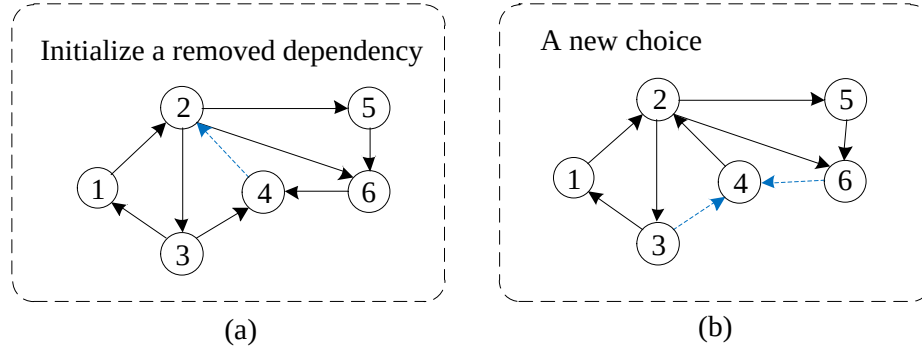


Figure 3.3: Example of Cycle Breaking

Figure 3.3 shows an example of cycle-breaking. In this example, the object relation diagram contains six nodes. We choose the edge $4 \rightarrow 2$ as the initially removed dependency in Figure 3.3(a), which breaks three cycles in the diagram: (1) $2 \rightarrow 3 \rightarrow 4 \rightarrow 2$, (2) $2 \rightarrow 6 \rightarrow 4 \rightarrow 2$, (3) $2 \rightarrow 5 \rightarrow 6 \rightarrow 4 \rightarrow 2$. A new choice with lower stubbing cost devised by our cycle-breaking algorithm are edges $3 \rightarrow 4$ and $6 \rightarrow 4$ in Figure 3.3(b). The edge $3 \rightarrow 4$ is removed to break the cycle (1), and edge $6 \rightarrow 4$ 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 1 is the main body of the proposed cycle-breaking algorithm ANS, 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 algorithm

Algorithm 1 Cycle-Breaking ANS Algorithm - Part I Main Body

Require: a list of edges that are involved in cycles *AllEdges*a list of cycles in the entire program *AllCycles***Ensure:** a set of removed edges *RemEdges*

```

1: RemEdges  $\leftarrow \emptyset$ 
2: for AllCycles.size  $\neq 0$  do
3:    $e = \text{getMaxCyclesEdge}(\text{AllEdges})$ 
4:    $scplx = \text{getSCplx}(e)$ 
5:    $ECycles = \text{getCycles}(e)$ 
6:    $EEdges = \text{getEdges}(ECycles)$ 
7:    $TempEdges1 \leftarrow \text{findFromCycles}(ECycles, scplx)$ 
8:    $TempEdges2 \leftarrow \text{findFromEdges}(EEdges, scplx)$ 
9:   if  $TempEdges1.size \neq 0 \parallel TempEdges2.size \neq 0$  then
10:     $RemEdges.add(\text{Compare}(TempEdges1, TempEdges2))$ 
11:   else
12:     $RemEdges.add(e)$ 
13:   end if
14:    $\text{update}(RemEdges)$ 
15: end for

```

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* (Algorithm 2, lines 1-12) and *findFromEdges* (Algorithm 2, lines 13-26). The two sets *TempEdges1* and *TempEdges2* store the set of removed edges generated by *findFromCycles* and *findFromEdges*, respectively.

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.

Algorithm 2 introduces these two functions *findFromCycles* and *findFromEdges* in detail. 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

3). If e_t is not equal to edge e and the overall stubbing cost after considering e_t is reduced (line 4), then e_t is added to the set $TempEdges1$, and consequently, all of the cycles in which e_t is involved are updated (lines 5-6). If the algorithm is unable to obtain better alternatives, it returns an empty set (line 8). After all of the cycles in set $ECycles$ have been processed, the final set $TempEdges1$ is obtained.

Algorithm 2 Cycle-Breaking ANS Algorithm - Part II Functions

```

1: function findFromCycles( $ECycles$ ,  $scplx$ )
2: for each cycle  $c$  in  $ECycles$  do
3:    $e_t = \text{getMinSCplxEdge}(c)$ 
4:   if  $scplx > \text{SCplx}(e_t, TempEdges1)$  then
5:      $TempEdges1.add(e_t)$ 
6:      $\text{update}(e_t)$ 
7:   else
8:      $TempEdges1 \leftarrow \Phi$ 
9:   break
10:  end if
11: end for
12: end function

13: function findFromEdges( $EEdges$ ,  $scplx$ )
14:  $num = \text{getCyclesNum}(e)$ 
15: for  $num > 0$  do
16:    $e_t = \text{getMaxCyclesEdge}(EEdges)$ 
17:   if  $scplx > \text{SCplx}(e_t, TempEdges2)$  then
18:      $TempEdges2.add(e_t)$ 
19:      $num = num - \text{getCyclesNum}(e_t)$ 
20:      $\text{updateEdges}(e_t)$ 
21:   else
22:      $TempEdges2 \leftarrow \Phi$ 
23:   break
24:   end if
25: end for
26: end function

```

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 16). If removing e_t rather than e can reduce the overall stubbing cost, then e_t is added to the set *TempEdges2* (lines 17-18). The number of remaining cycles is updated (line 19) as are the other edges in the set *EEdges* (line 20). Similarly, an empty set is returned when no better alternatives exist (line 22). After the set *TempEdges2* breaks the same number of cycles as the initial edge e , the execution of this function is completed.

For the time complexity of these two algorithms (Algorithm 1 and 2), 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, 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 CITO to generate an offspring. Therefore, the proposed measure for graph-based methods cannot be applied to search-based methods.

3.5 Experimental Design

We designed two groups of experiments to answer the following research question:

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

To answer this question, 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 OCplx is calculated by the Equation 3.2. 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.

We first used benchmark programs to compare ANS algorithm with two typical methods. That is, the graph-based cycle-breaking algorithm [12] and a search-based method that adopted genetic algorithm [11] proposed by Briand et al., which are commonly used as competitors by other researchers. Table 3.2 shows the information of benchmark programs, including their descriptions and number of classes.

Table 3.2: Information of Used Programs in Group 1

Programs	Description	Classes
ANT	A Java-Based Build Tool	25
ATM	Automated Teller Machine Simulation System	21
DNS	Domain Name System	61
SPM	Security Patrol Monitoring System	19

To further evaluate the performance of the proposed approach, we selected nine Java programs that are widely used in the existing CITO generation approaches. Table 3.3 summarizes these programs, including their sources, descriptions, and numbers of classes. Three cycle-breaking strategies mentioned in Section 3.3 are used as competitors to validate the effectiveness of the proposed algorithm, ANS. Because the source code of the

three cycle-breaking strategies are not available, we implemented them according to their algorithms.

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.

Table 3.3: Information of Used Programs in Group 2

Programs	Source	Description	Classes
daisy	https://sir.csc.ncsu.edu/php/common/download.php?ac=pub&key=sir/javaobjects&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/javaobjects&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/javaobjects&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/javaobjects&id=158&file=GPL__spl_1.0.tar.gz (v1.0)	Graph generator	178
JHotDraw	http://sourceforge.net/projects/jhotdraw/packageused: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/javaobjects&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/javaobjects&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/javaobjects&id=162&file=notepad__spl_1.0.tar.gz (v1.0)	Source code editor	65

For benchmark programs, information about class dependencies including attribute coupling and method coupling was taken from the Appendix in the literature [11]. For the programs in Group 2, the class dependencies and the related coupling information were obtained using a Java program analysis framework called Soot¹.

3.6 Experimental Results

Table 3.4 shows the comparison of results for four benchmark programs. The values of two quality indicators (OCplx and Stubs) for these two methods were obtained from reference

¹<http://www.sable.mcgill.ca/soot>

[11]. The values of OCplx in Briand et al.'s approach (graph-based method) is null since it only considers the number of test stubs. GA refers to the CITO generation approach based on genetic algorithm (GA). We do not conduct statistic test for benchmark programs since we do not have the raw data of comparative approaches. The best mean values of two quality factors for each program has been highlighted in bold.

In terms of OCplx, the mean values obtained by ANS are minimal for two programs compared with GA. For the program *ANT*, although the mean values for both GA and ANS are 3.62, GA is non-determinism whose OCplx ranges from 3.59 to 3.86. In terms of Stubs, ANS obtains the minimal values on three programs. Even for the remaining program *ANT*, ANS only requires one more test stub than the best solution.

Table 3.4: Comparison of Results on Benchmark Programs for ANS and Briand et al.'s methods

Programs	Methods	OCplx		Stubs	
		Range	Mean	Range	Mean
ANT	Briand et al.	-	-	11	11
	GA	[3.59-3.86]	3.62	[12-14]	12.16
	ANS	3.62	3.62	12	12
ATM	Briand et al.	-	-	7	7
	GA	2.68	2.68	7	7
	ANS	2.70	2.70	7	7
DNS	Briand et al.	-	-	6	6
	GA	1.47	1.47	[6-7]	6.89
	ANS	1.00	1.00	5	5
SPM	Briand et al.	-	-	17	17
	GA	[5.77-6.52]	5.87	[16-18]	16.94
	ANS	2.45	2.45	11	11

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 3.3. 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 [38] 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 [12]. The ranges, means and p -values of OCplx and Stubs are listed in Table 3.5, 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 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

Table 3.5: Comparison of Results on Programs for ANS and Three Competitors (NC, SCplx and CWR)

Programs	Methods	OCplx			Stubs		
		Range	Mean	<i>p</i> -value	Range	Mean	<i>p</i> -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	-

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.

3.7 Threats to Validity

Although the experimental results demonstrate the effectiveness of ANS algorithm, it still faces the following threat to 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.

3.8 Conclusion

The number of test stubs and the corresponding stubbing complexity are two main factors that affect the overall stubbing cost for a CITO. Most of the existing techniques for CITO generation ignore these two factors, or unable to fully consider these two factors.

Therefore, we proposed a new cycle-breaking algorithm, ANS. The basic idea of ANS is to perform a good Analysis of the relationship between the Number of removed dependencies and the related Stubbing complexity. ANS first selects a class dependency as an initialization. Then, it searches 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 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.

Chapter 4

CITOSC: Evaluating the Effects of Similar-Class Combination on Class Integration Test Order Generation

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.

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), therefore, the main objective of this study is reducing the problem space for CITO generation.

We proposed a method called **CITOSC** (Class Integration Test Order generation with Similar-class Combination), which 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. After identifying similar classes, an algorithm is proposed to reduce the

problem space by combining such similar classes. In addition, we combine the similar-class combination with the novel cycle-breaking algorithm ANS proposed in the previous chapter. Empirical experiments are conducted on nine open-source Java programs to evaluate the performance of the proposed approach.

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 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. Moreover, after combining similar classes, our cycle-breaking algorithm ANS still outperformed other competing methods in minimizing the stubbing costs of the generated test orders.

The rest of this chapter is organized as follows. Section 4.1 introduces the whole work of this chapter. Section 4.2 introduces the motivation. We present our approach in Section 4.3. Section 4.4 presents the detailed design of our experiments and Section 4.5 summarizes the results. Section 4.6 discusses our findings and the possible threats to validity. Section 4.7 concludes this chapter.

4.1 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 [117]. 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. Especially in large-scale programs, stubbing is inevitable because circular dependencies are quite common [86].

Designing an optimal test order with the minimal stubbing cost is extremely important. However, searching for the globally optimal class test order is impractical, even when the number of classes is moderately large. Graph-based and search-based approaches have been extensively adopted to search for the locally optimal class test order instead [5]. 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 the following problem:

- It becomes difficult for the existing heuristics to generate the locally optimal test order for real applications with a moderately large number of classes.

This 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 [12] 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.

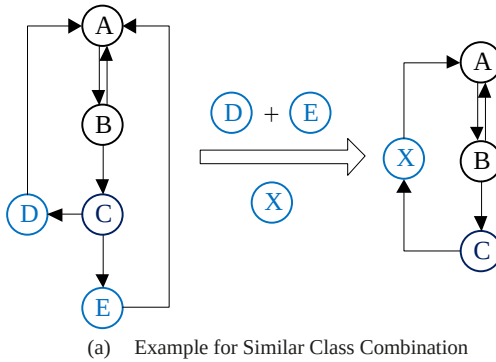
We aim to reduce the problem space to address this issue. The CITO generation is basically equivalent to the problem of the identification of a minimal feedback vertex set (FVS) [79]. To simplify the problem of finding an FVS, Orenstein et al. [94] proposed a new kind of graph reduction operation in 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.

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%. We also evaluate the effects of similar-class combination on the performance of the state-of-the-art cycle-breaking techniques, including the ANS algorithm proposed in the previous chapter. Overall, similar-class combination does not degrade the performance of the cycle-breaking techniques when considering the coupling values of class dependencies.

4.2 Motivation

In this section, we will use a sample program to illustrate why the similar-class combination is able to reduce the problem space for the CITO generation methods. Then, we present our approach to address this problem.



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

Figure 4.1: Example for Similar Class Combination

We aim to combine similar classes to reduce the problem space for the existing approaches. Figure 4.1 shows an example of similar-class combination for the sample program in Figure 3.2. As shown in Figure 3.2(a), which is the same as the Object Relation Diagram (ORD) shown on the left side of Figure 4.1(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 Figure 4.1(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 Figure 4.1(b).

After combining similar classes, the class dependency $B \rightarrow A$ owns the minimal stubbing complexity and the greatest cycles-weight ratio, rather than the class dependency $C \rightarrow D$ in the original program in Figure 3.2. Corresponding, the class dependency $B \rightarrow A$ will be removed by two cycle-breaking algorithms mentioned in the Section 3.3, SCplx and CWR. Finally, SCplx and CWR can also generate the optimal solution for the reduced program.

4.3 Proposed Approach

This section introduces the proposed CITO generation approach. We first illustrate the overview of the approach, then introduce the two properties for similar-class identification, and finally describe our similar-class combination algorithm.

4.3.1 Overview of the Approach

Figure 4.2 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. 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

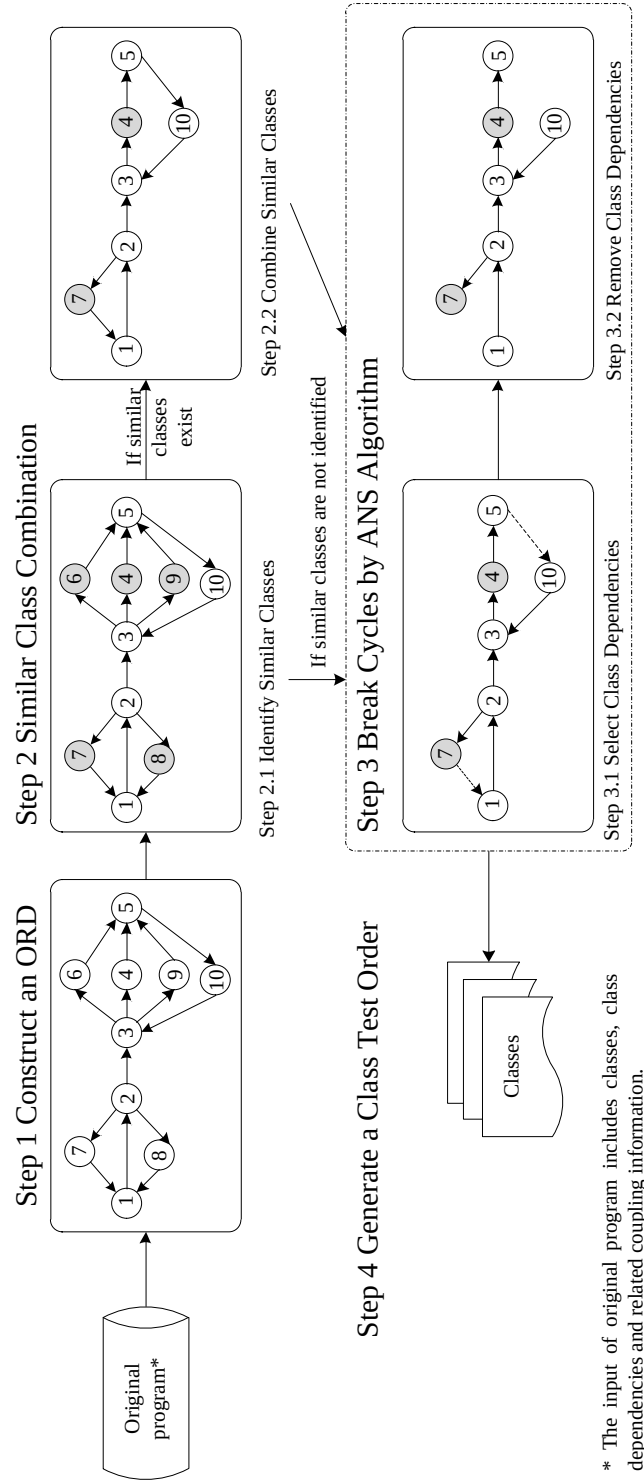


Figure 4.2: Overview of the Proposed Approach

removed. Thereafter, a "reduced" graph that contains fewer class dependencies and class cycles is generated.

Next, Tarjan's algorithm [119] is adopted to divide the diagram into different strongly connected components. We identify all of the cycles in each strongly connected component.

The cycle-breaking algorithm **ANS**, which aims to keep a good Analysis involving the Number of removed dependencies and the related Stubbing complexity, is applied to each strongly connected component. 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.

4.3.2 Properties for Similar-Class Identification

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 chapter, 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., $Target_C_i = Target_C_j \wedge Source_C_i = Source_C_j$, 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, $Cycles_C_i \not\subseteq Cycles_C_j$ and $Cycles_C_j \not\subseteq Cycles_C_i$ 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 $c_j = C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_n \rightarrow C_j \rightarrow C_1$ ($1 \leq n \leq m$, m is the number of classes) belongs to $Cycles_C_j$

whereas the cycle $c_i = C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_n \rightarrow C_i \rightarrow C_1$ 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 $p = C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_n$ 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 $c_j = C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_n \rightarrow C_j \rightarrow C_1$ belongs to $Cycles_{C_j}$ whereas the cycle $c_i = C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_n \rightarrow C_i \rightarrow C_1$ 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 $c_i = C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_n \rightarrow C_i \rightarrow C_1$ ($1 \leq n \leq m$, m is the number of classes) in $Cycles_C_i$, there is a corresponding cycle $c_j = C_1 \rightarrow C_2 \rightarrow \dots \rightarrow C_n \rightarrow C_j \rightarrow C_1$ 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.

4.3.3 Similar-Class Combination

Similar-class combination is inspired by the graph reduction operation proposed by Orenstein et al. [94] 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 CITO. 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 Figure 4.1 introduced in Section 4.2, 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 4.6 with the aid of real applications.

Combining symmetric classes does not confront with the abovementioned 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 3 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).

Algorithm 3 Similar Class Combination

Require: a set of all classes $Classes$

a set of all cycles $Cycles$

Ensure: 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

```

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 $C_1, C_2, \dots, C_n$. 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 Equations 4.1 and 4.2, 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.

$$SCplx(S, C) = \sum_{C_i \in sim(C)} SCplx(S, C_i) \quad (4.1)$$

$$SCplx(C, T) = \sum_{C_i \in sim(C)} SCplx(C_i, T) \quad (4.2)$$

4.4 Experimental Design

We conducted several experiments to evaluate the performance of the proposed approach. Research questions are introduced in Section 4.4.1. The information of used programs, experimental settings, and evaluation metrics are described in Section 4.4.2.

4.4.1 Research Questions

The proposed 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 two research questions. RQ1 pertains to the effectiveness of the similar-class combination under the condition that the other factors remain unchanged. RQ2 assesses the performance of the proposed approach (i.e., ANS algorithm with similar-class combination) 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 CITO, 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 4.3.2, 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: 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 three kinds of cycle-breaking algorithms mentioned in Section 3.3 to the reduced programs obtained by similar-class combination and compare the results.

4.4.2 Experimental Settings and Evaluation Metrics

To be consistent with experimental settings in Chapter 3, the nine Java programs in Table 3.3 were also used in this experiment. Therefore, it would be easy to compare the performance of the cycle-breaking techniques before and after applying similar-class combination.

Similarly, the overall stubbing complexity (OCplx) was used to evaluate the stubbing cost of constructing test stubs based on generated class test orders. The number of test stubs (Stubs) was considered as a supplementary metric. The lower values of these two metrics indicate a better performance of the CITO generation approach.

4.5 Experimental Results

4.5.1 Research Question 1

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 4.1 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. Figure 4.3 shows the relationship between

Table 4.1: 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

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 4.2 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 compared with the original programs. 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%

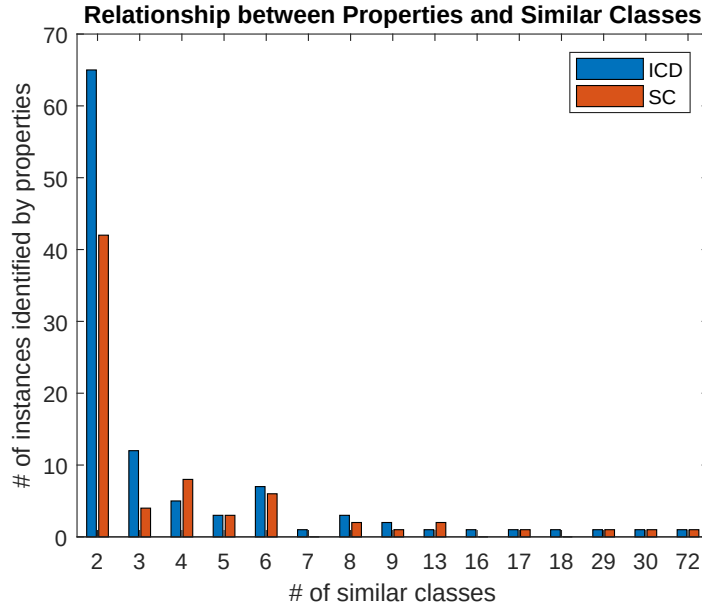


Figure 4.3: Relationship between Properties and Similar Classes

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. Figure 4.4 shows a comparison of two ORDs generated for the program *notepad__spl*. Due to space limitations in this thesis, the diagrams of the other programs can be accessed on GitHub (https://github.com/miazhang9/CITO_SCC). Figure 4.4(a) shows the ORD for the original program, where the value in each node indicates the

Table 4.2: Comparison of Programs Obtained by Combining Similar Classes based on Two Properties

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%

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 Figure 4.4(b). Two types of similar classes that are shown in Figure 4.4 were identified by both properties, and the class numbers of these classes are shown as follows:

- 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 Figure 4.4(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 Figure 4.4(b). The new program contains only 30 cycles compared with 227 cycles in the original program.

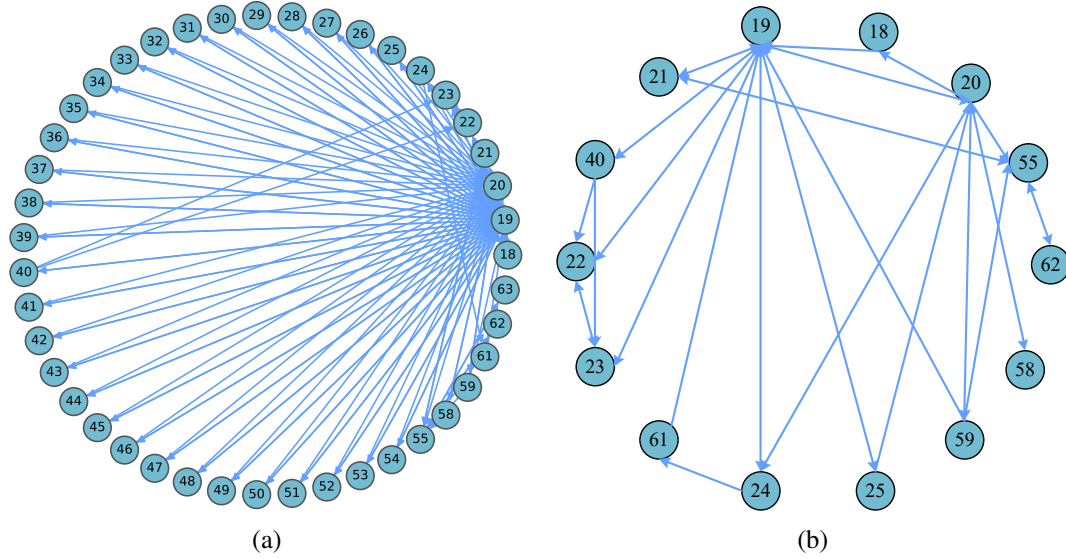


Figure 4.4: Comparison of Object Relation Diagram for `notepad__spl`: (a) Original Program (b) Reduced Program

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 CITO, as mentioned in Section 4.1. 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.

4.5.2 Research Question 2

RQ2: Effects of similar-class combination on the cycle-breaking techniques.

To answer RQ2, we evaluate the effects of similar-class combination on the performance of the cycle-breaking techniques, and this question includes two research sub-questions:

- RQ2.1 What is the performance of the proposed cycle-breaking algorithm (ANS) in minimizing the stubbing cost when applying similar-class combination?
- RQ2.2 Will similar-class combination affect the performance of other cycle-breaking techniques?

Tables 4.3 and 4.4 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. Values in Tables 4.3 and 4.4 that are different from those in Table 3.5 are highlighted in bold.

As in the case of Research Question in Chapter 3, 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 RQ2.1, as shown in Table 4.3, the proposed approach achieved the minimal stubbing cost for six of the eight programs (except for *log4j3* and *MyBatis*). The results shown in Table 4.4 are identical to our findings for Research Question in Chapter 3: the proposed approach outperformed the competitor approaches for all programs, except *MyBatis*.

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 Research Question in Chapter 3.

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

Table 4.3: Comparison of Results on Programs After Combining Similar Classes by Property ICD

Programs	Methods	OCplx			Stubs		
		Range	Mean	<i>p</i> -value	Range	Mean	<i>p</i> -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	-

Table 4.4: Comparison of Results on Programs After Combining Similar Classes by Property SC

Programs	Methods	OCplx			Stubs		
		Range	Mean	<i>p</i> -value	Range	Mean	<i>p</i> -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	-

(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 $16 \rightarrow 18$ instead of $36 \rightarrow 16$ and $38 \rightarrow 16$, as selected by CWR. Table 4.5 shows the information of these three class dependencies; the class dependency $36 \rightarrow 16$ has a higher cycles-weight ratio than the other class dependencies. Therefore, the class dependency $36 \rightarrow 16$ was first removed by CWR. Then, the cycles-weight ratio was updated for the remaining class dependencies. The class dependency $16 \rightarrow 18$ was involved in eight cycles, where two cycles also contained the class dependency $36 \rightarrow 16$. Therefore, its cycles-weight ratio was updated to 61.224 ($6/0.098=61.224$). The class dependency $38 \rightarrow 16$ was still involved in two cycles, so it yielded the highest cycles-weight ratio (62.500). Therefore, CWR removed the class dependency $38 \rightarrow 16$. 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 $X \rightarrow 16$ ($2/(0.011+0.032) = 46.512$). Correspondingly, the class dependency $16 \rightarrow 18$ 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 $16 \rightarrow 18$ led to the greatest increase in stubbing cost by far, CWR-ICD eventually yielded a better result.

Table 4.5: Class Dependencies Causing Better Result of CWR-ICD for Program log4j3

Class Deps	# of Cycles	Stubbing Complexity	Cycles-Weight Ratio
$36 \rightarrow 16$	2	0.011	181.818
$16 \rightarrow 18$	8	0.098	81.633
$38 \rightarrow 16$	2	0.032	62.500

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 Research Question in Chapter 3. 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 RQ2.2, In Table 4.3 and 4.4, values of OCplx and Stubs that are different from those in Table 3.5 are presented in boldface for the sake of readability, while the best values of OCplx and Stubs are presented in boldface in Table 3.5.

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 4.6 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.

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*.

Table 4.6: 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
JHDraw	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

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 4.6.

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.

4.6 Discussion

4.6.1 Analysis for Special Cases

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 CITOes 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 4.7 presents the programs for which the generated CITOes 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

Table 4.7: Programs for Which Generated CITOes Are Affected by Similar Class Combination

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(↓)

MyBatis), which is consistent with the explanation given in Section 4.3.3. Although the 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 $20 \rightarrow 19$ (stubbing complexity = 0.713) involved in 30 cycles was removed by NC in a certain stage of its execution, as shown in Figure 4.5. Both ICD and SC identified 29 similar classes in these cycles, which were highlighted in grey background. We regarded these 29 classes holistically, and constructed a new class dependency $19 \rightarrow 25$ (stubbing complexity = $0.019 \times 29 = 0.551$) to represent all dependencies from class 19 towards these similar classes. After similar-class combination, class dependency $19 \rightarrow 25$ was involved in seven cycles (the actual number of cycles should be $29 \times 7 = 203$), which was the greatest number of cycles. While the class dependency $20 \rightarrow 19$ was involved in only two cycles. Hence, NC-ICD and NC-SC removed class dependency $19 \rightarrow 25$ rather than $20 \rightarrow 19$, breaking 173 more cycles with 0.162 less stubbing complexity.

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;

Figure 4.5: Cycles Containing Class Dependency $20 \rightarrow 19$

The proposed approach yielded fewer OCplx and Stubs for the reduced program *notepad__spl* because of the removal of an extra class dependency $18 \rightarrow 19$ from the original program. As we observed in the case of Research Question in Chapter 3, 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 Figure 4.4, 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 $51 \rightarrow 182$, $51 \rightarrow 186$, $51 \rightarrow 192$, and $51 \rightarrow 193$ 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.

4.6.2 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. 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. [12], Hashim et al. [55], and Abdurazik et al. [3], 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. [12] 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. [55] 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. [3] regards the number of parameters and return value types as two different coupling values, but we combine them into attribute coupling in this thesis, as described in Section 3.2. 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.

4.7 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, the cycle-breaking algorithm ANS in Chapter 3 was combined 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 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 section, 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.

Chapter 5

CITOMT: Validating Class Integration Test Order Generation Systems with Metamorphic Testing

Previous studies proposed different kinds of approaches for class integration test order (CITO) generation, and corresponding systems can be implemented based on these approaches. Such CITO generation systems can facilitate the process of software integration testing if they are implemented correctly.

However, a test oracle problem exists in the CITO generation systems. Since these approaches for CITO generation normally deliver a local optimum rather than a global optimum, there are no practically feasible ways to validate their generated CITO, that is, these implementation systems are untestable.

To address the test oracle problem, we apply Metamorphic Testing (**MT**) to validate **CITO** generation systems and implement it as **CITOMT**. Metamorphic Relations (MRs), which are the key components of MT, reason about relations between test outputs of a

system. Five effective MRs are developed to ensure the quality of the CITO generation systems. In these proposed MRs, follow-up test inputs are generated by modifying classes or class dependencies in the source test inputs while some characteristics of the source test outputs are preserved, for example, the same class test order or the equal stubbing cost. Faults can be detected in systems if an individual MR is violated for certain tests.

Failure detection of MT has been successfully demonstrated in empirical experiments on three systems implementing different typical CITO generation approaches. More than 84% of faulty programs can be detected by all MRs, for three CITO generation systems under investigation. The experimental results show that the proposed MRs are able to systematically and effectively detect faults in CITO generation systems. This study explores a new application domain in MT and further extends its applications in Software Engineering.

The rest of this chapter is organized as follows. Section 5.1 introduces the whole work of this chapter. We present our validation framework in Section 5.2. Section 5.3 presents the detailed design of our experiments and Section 5.4 summarizes the results. Section 5.5 discusses our findings and the possible threats to validity. Section 5.6 concludes this chapter.

5.1 Introduction

Software integration testing is an essential process to guarantee the quality of software being produced. An important problem in software integration testing is to determine the order in which classes are integrated and tested, because classes in a software system are usually independently developed and tested. Test stubs are constructed for a class *A* when

classes on which class *A* depends are unavailable. Because cycles among class dependencies are common in programs [86], test stubs are inevitable.

Different class integration test orders (CITOs) need various test stubs, and correspondingly, require different efforts to construct such test stubs, which has been proved by Kung et al. [67]. In their experiment, students were asked to prepare test stubs for some member functions for a program, and one simple test stub is required about 0.79 person-hours to prepare. For a case study system called InterViews which contains 122 classes, a random CITO requires 191 test stubs that could be written in about 152 person-hours, while the optimal CITO requires only 8 test stubs that can be constructed in around 7 person-hours. Therefore, for an ideal CITO, few or no efforts required to construct test stubs for the integration testing are desirable.

Finding the optimal CITO is basically equivalent to the travelling salesman problem [12], which is well-known as a factorial ordered problem. Searching for the globally optimal CITO is impractical for a real-life application even with a moderately large number of classes, and hence heuristics are extensively adopted to search for the locally optimal CITO instead.

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

However, testing such a system is difficult if not impossible, because except for trivial inputs, no practically feasible ways exist to validate its generated CITO. Approaches for

CITO generation would be more likely to deliver a locally optimal solution than a globally optimal solution. Given a real-life system of even moderate complexity, we are not able to validate whether a generated CITO for this real-life system is the actual output that should be computed by the approach, as we are unable to give an exact definition for the concept of local optimum. Overall, such CITO generation systems are referred to as untestable systems or systems with the test oracle problem.

Fortunately, the technique of metamorphic testing (MT) [26] could help to alleviate the test oracle problem. MT differs from conventional testing approaches which focus on the verification of each individual output of the software under testing, instead MT verifies relations among the inputs and outputs of multiple executions of the software. Such relations are referred to as metamorphic relations or MRs for short. If an individual MR is violated for certain tests, it implies that the implementation is faulty because MRs are the necessary properties of the implemented algorithm. In this case, the test oracle is not required, which is suitable for our untestable systems.

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

In this chapter, we propose five kinds of MRs to validate different CITO generation systems. These MRs modify class dependencies with the aim of preserving a part of or the entire original test order. We validate three systems implementing different typical CITO generation approaches using the test suites generated based on the proposed MRs. The results show that no faults can be detected by the five proposed MRs in three CITO generation systems, which increases the likelihood of being valid for our systems. We

also conduct mutation analysis to evaluate the effectiveness of the proposed MRs. The experimental results show that more than 84% of faulty programs can be detected by all MRs for three CITO generation systems.

5.2 Validation Framework

5.2.1 Validation Procedure

In this section, we adopt MT to validate systems implementing the CITO generation approaches introduced in Section 3.3. MR is the key component in MT, which is a necessary property of the function over its inputs and the corresponding outputs. MR describes how to generate a follow-up test case based on the source test case and its source output, and explains which relation between two outputs (i.e., source output and follow-up output) must be satisfied. We first extract MRs for these systems implementing the CITO generation approaches. The proposed MRs are generated based on the necessary property of these approaches, which will be explained in detail in Section 5.2.2. The main procedure of these CITO generation approaches is identical, which is to break all cycles in the object relation diagram constructed for the program under integration testing. Once no cycle exists in the diagram, classes are integrated and tested based on the reverse topological sorting. Test stubs are only constructed for the class dependencies whose corresponding edges are removed in cycles.

Our validation procedure for the CITO generation systems includes four steps. Firstly, we select source programs as source inputs to run the CITO generation system and obtain the generated test orders as source outputs. Then, follow-up programs are generated based on the proposed MR, source inputs and their corresponding source outputs. Such follow-up programs are referred to as follow-up inputs. Next, follow-up inputs are used to run the

CITO generation system. The obtained test orders are follow-up outputs. Finally, we check whether the proposed MR holds for such two kinds of outputs. If the relation is violated, a failure is revealed.

5.2.2 Metamorphic Relations (MRs)

In this section, we discuss the developed MRs for the CITO generation systems.

MR1: Permutation of class names

An obvious MR to validate CITO generation systems is about permuting class names. Figure 5.1 shows an example for the permutation of class names. In the source program and follow-up program, each node represents a class and each directed edge represents a class dependency, for example, $C_1 \rightarrow C_2$ represents class C_1 depends on class C_2 . Let SP denote the source program that contains *six* classes, and the output is a class integration test order, O_{sp} , which is further supposed to be $[C_4, C_5, C_3, C_2, C_1, C_6]$. A follow-up program FP can be constructed by permuting the class names without any modifications on inter-class relationships and functionality of these classes. Obviously, there is a one-to-one correspondence between the computed test orders of programs SP and FP , and the efforts required to construct test stubs based on these two test orders should also be equal.

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

Figure 5.2 shows an example for MR2. The notation for nodes and edges is similar with Figure 5.1, and the weight for each edge represents the stubbing cost for each class dependency. Let SP denote the source program that contains *six* classes, and in the output, i.e., the generated class integration test order, O_{sp} , class C_4 is integrated and tested first.

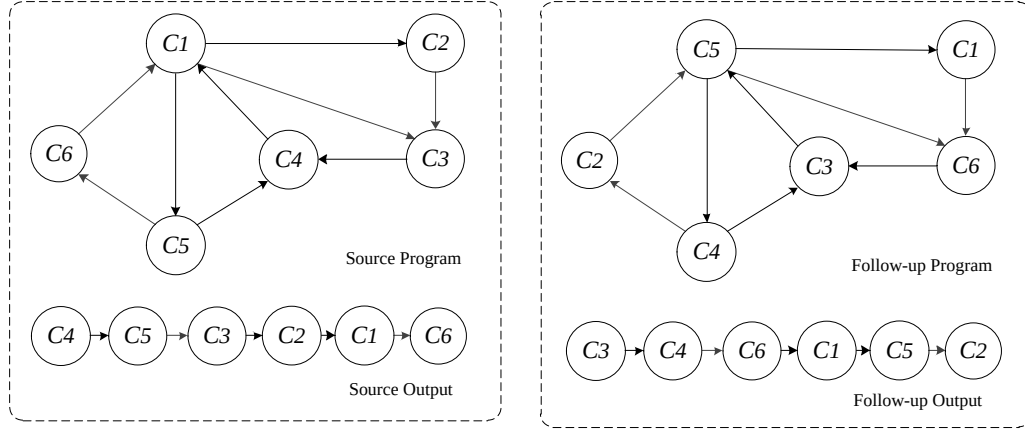


Figure 5.1: Example of MR1 Permutation of Class Names*

*The adopted permutation scheme is $C_1 \rightarrow C_5$, $C_2 \rightarrow C_1$, $C_3 \rightarrow C_6$, $C_4 \rightarrow C_3$, $C_5 \rightarrow C_4$, $C_6 \rightarrow C_2$.

It indicates that the highest test priority of class C_4 will contribute to a lower stubbing cost for the subsequent integration testing (i.e., test stubs are not required for those classes depending on class C_4). In such a case, adding new class dependencies to class C_4 will not incur extra stubbing cost for the subsequent integration testing, and this operation will not affect the test order of class C_4 . Hence, a follow-up program FP which has the same classes as SP , can be generated by adding one or more class dependencies to class C_4 , for example, the addition of edge $C_2 \rightarrow C_4$. Class C_4 will still be the initial class in the test order O_{fp} generated for the program FP . Because the classes on which class C_4 depends are unchanged, the stubbing cost of O_{fp} will be equal to that of O_{sp} . Similarly, deleting class dependencies from class C_4 , will reduce the stubbing cost when class C_4 is integrated, and as a consequence, this operation will not affect its test priority. Therefore, the other follow-up program FP which has the same classes as SP , can be constructed by removing class dependencies through the removal of edges, such as, removing class C_1 on which class C_4 depends, through the removal of edge $C_4 \rightarrow C_1$. Class C_4 will still be the initial class in

the class integration test order O_{fp} generated for the program FP . Because the classes on which class C_4 depends are fewer in FP , the stubbing cost of O_{fp} will be smaller than that of O_{sp} .

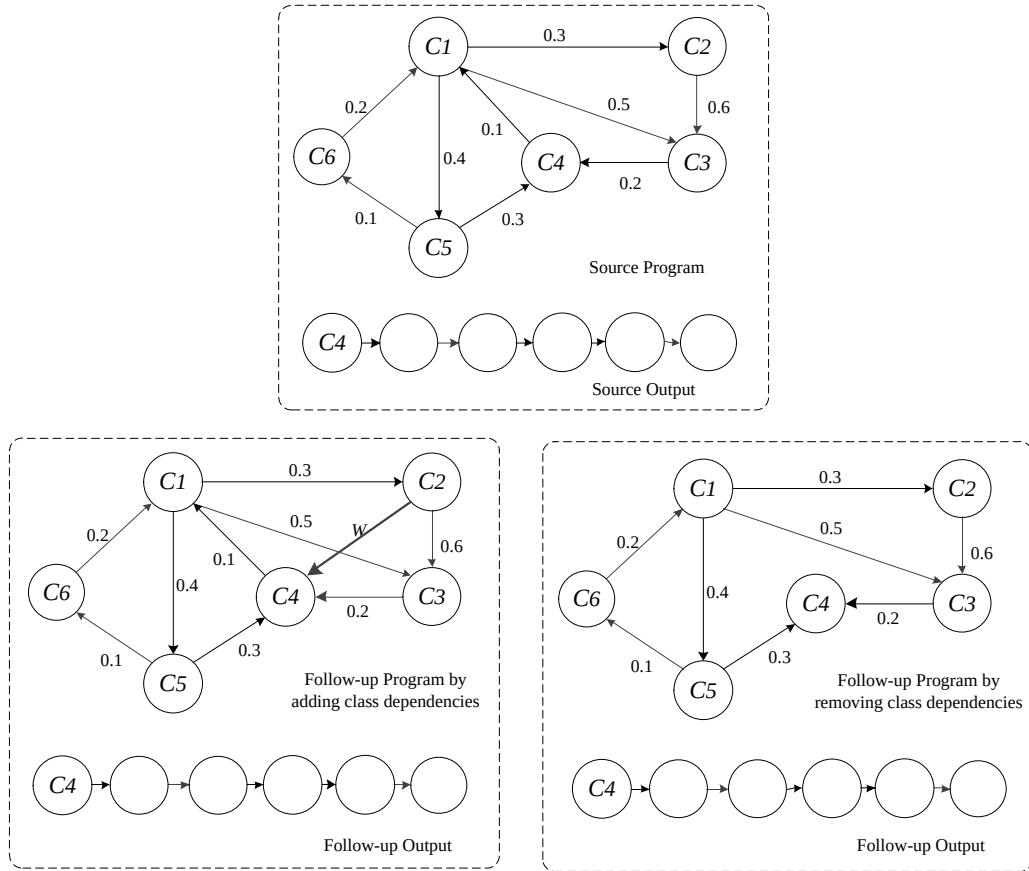


Figure 5.2: Example of MR2 Adding/deleting class dependencies for the initial class*

*The added class dependency is $C_2 \rightarrow C_4$ with a very large weight W , and the removed class dependency is $C_4 \rightarrow C_1$.

To prove MR2, we need the following proposition:

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

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

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

Another measure to generate the follow-up program is to add new class dependencies to class C . If such class dependencies do not form new cycles, the locally optimal solution obtained by class integration test order generation approaches will be the same, and class C will still be the initial class in the integration test order. If the newly added class

dependencies form new cycles, some class dependencies from class C must be involved in these cycles. Removing such class dependencies from class C will be the optimal solution to break these cycles if they own lower stubbing cost. Therefore, to make sure that class C will still be the initial class in the class integration test order, the newly added class dependencies should be assigned a very large weight W .

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

Figure 5.3 shows an example for MR3, where the notation for nodes and edges is similar with that of other MRs. Let SP denote the source program that contains *six* classes, and in its generated class integration test order, O_{sp} , class C_6 is the last class to be integrated and tested. That means testing class C_6 requires the greatest stubbing cost, or this operation cannot lower stubbing cost for the subsequent integration testing (i.e., fewer classes depending on class C_6). A MR can be defined relating to the modification of class dependencies for the last class in the integration test order, i.e., class C_6 . Because testing class C_6 has already incurred the greatest stubbing cost, the increase of its stubbing cost by adding more class dependencies from class C_6 will not affect the test order of class C_6 . Hence, a follow-up program FP which has the same classes as SP , can be produced by adding one or more dependencies from class C_6 to other classes, for example, adding edge $C_6 \rightarrow C_2$. Class C_6 will still be the last class in the class integration test order O_{fp} generated for the program FP . Because class C_2 is integrated and tested prior to class C_6 , no extra test stubs are required in the program FP . The stubbing cost of O_{fp} will be the same as that of O_{sp} . Similarly, reducing the stubbing cost saved by the integration of class C_6 for the subsequent integration testing has no effects on the test order of class C_6 . Therefore, suppose that class C_6 is depended by some other classes, a follow-up program FP which has the same classes as SP , can be generated by removing one or more class dependencies to class C_6 , such as,

removing edge $C_5 \rightarrow C_6$. Class C_6 will still be the last class in the test order O_{fp} generated for the program FP . The test stub constructed for class C_5 to emulate the services from class C_6 is unnecessary now. The stubbing cost of O_{fp} will be lower than that of O_{sp} .

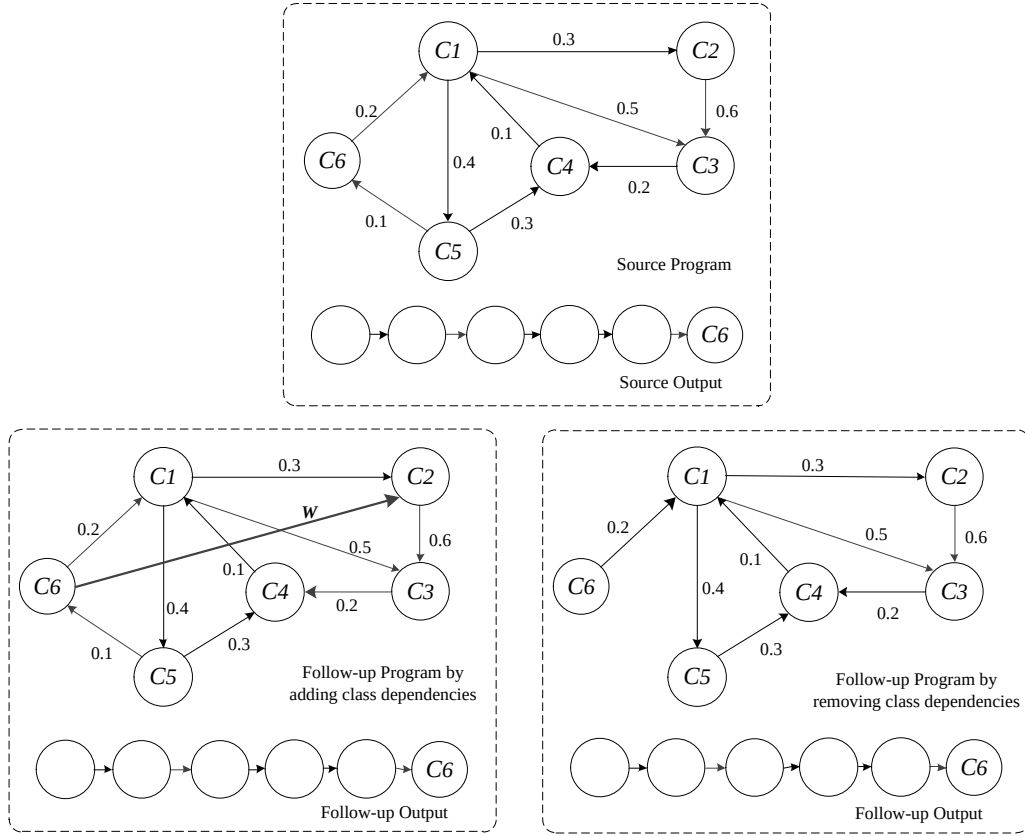


Figure 5.3: Example of MR3 Adding/deleting class dependencies for the last class*

*The added class dependency is $C_6 \rightarrow C_2$ with a very large weight W , and the removed class dependency is $C_5 \rightarrow C_6$.

To prove MR3, we need the following proposition:

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

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

Proof of MR3. Class C is integrated and tested lastly, which means all class dependencies to class C are the locally optimal solution obtained by class integration test order generation approaches to break cycles in which they are involved. Removing some class dependencies to class C breaks their corresponding cycles and does not affect the solutions to break the remaining cycles. Therefore, test stubs will be constructed for other remaining class dependencies to class C , and class C will still be the last class in the class integration test order.

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

MR4: Adding a new class with the aim of preserving the original test order

Different from MR2 and MR3 in which classes in the program remain unchanged, a new class is added to the program with the aim of preserving the original test order in MR4. Figure 5.4 shows an example for MR4. Let SP denote the source program that

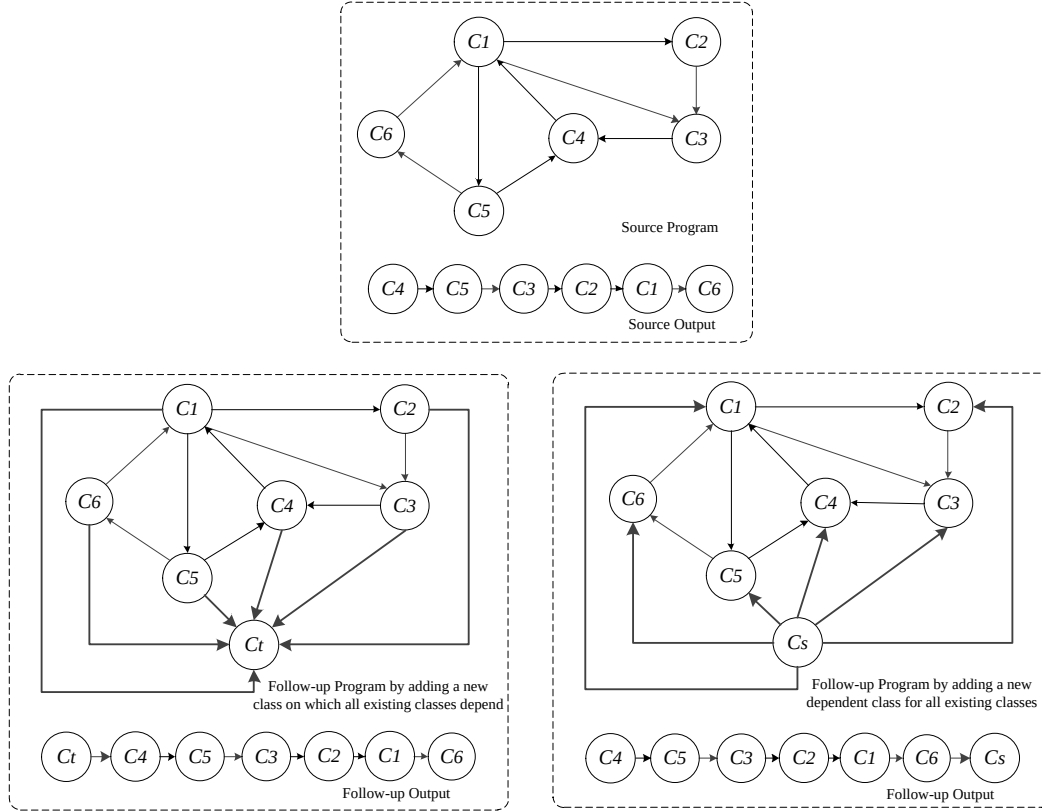


Figure 5.4: Example of MR4 Adding a new class with the aim of preserving the original test order*

* One kind of added class is C_t , which is depended by all the existing classes, and the other kind is C_s , which depends on all other existing classes.

contains *six* classes, and the class integration test order generated for the program SP is O_{sp} , which is further supposed to be $[C_4, C_5, C_3, C_2, C_1, C_6]$. A follow-up program FP can be generated by adding a new class C_t such that all the classes in the program depend on this new class C_t . Therefore, the class integration test order O_{fp} generated for the program FP should be $O_{fp} = [C_t, O_{sp}]$. Similarly, a new class C_s that depends on all other existing

classes is added, and the class integration test order O_{fp} should be $[O_{sp}, C_s]$. The stubbing cost of these two class integration test orders O_{fp} will be the same as that of O_{sp} .

MR5: Combining classes with the aim of preserving the original test order

A more complicated MR is to combine some classes with the aim of preserving the original test order. Figure 5.5 shows an example for MR5. Let SP denote the source program that contains *six* classes, and the class integration test order generated for the program SP is O_{sp} , which is further supposed to be $[C_4, C_5, C_3, C_2, C_1, C_6]$. Two edges $C_4 \rightarrow C_1$ and $C_5 \rightarrow C_6$ are removed to break cycles, and test stubs are constructed for the two corresponding class dependencies. A follow-up program FP can be generated by combining classes, for example, combining classes C_2 and C_3 to form a new class C_x . Dependencies between class C_2 and C_3 are ignored, while dependencies between class C_2 (or C_3) and other remaining classes are generated for the new class C_x . For instance, an edge $C_1 \rightarrow C_x$ with the weight as 0.8 replaces the edge $C_1 \rightarrow C_2$ (weight = 0.3) and the edge $C_1 \rightarrow C_3$ (weight = 0.5). In the follow-up program, the combination of classes C_2 and C_3 does not affect the class integration test order generated by approaches, therefore, the class integration test order generated for the program FP is similar with O_{sp} , that is, $O_{fp} = [C_4, C_5, C_x, C_1, C_6]$. The stubbing cost of O_{fp} will be the same as that of O_{sp} .

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

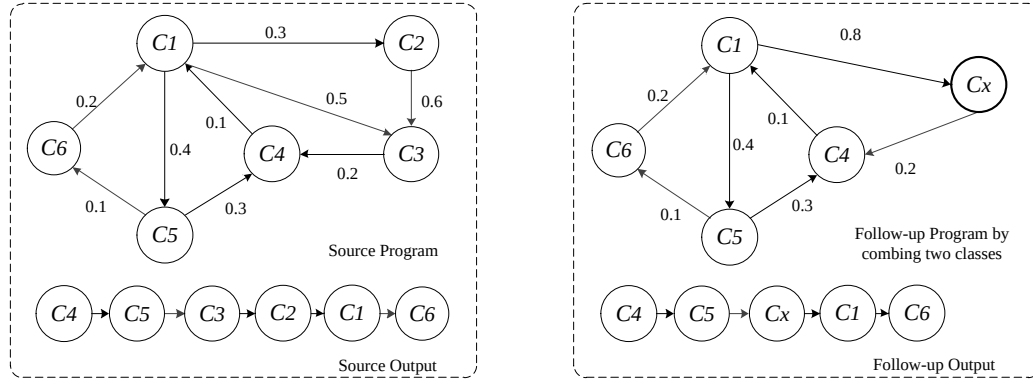


Figure 5.5: Example of MR5 Combining classes while preserving the original test order*

* Classes C_2 and C_3 are combined as a new class C_x .

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

5.3 Experimental Design

This section provides the experimental design for the proposed validation framework for three CITO generation systems. Test data, including source test cases and follow-up test cases are generated based on the proposed MRs. Firstly, source test cases and follow-up test cases are applied in three original CITO generation systems to detect any violations of these proposed MRs. If any violation is detected, which means that faults exist, then we will modify the systems to correct these faults. After that, mutation analysis is adopted to evaluate the effectiveness of five proposed MRs. Mutants are generated for the CITO

generation system as faulty programs. If an MR is violated, a mutant is killed and correspondingly, a fault can be detected. Section 5.3.1 introduces the information of three CITO generation systems. Section 5.3.2 introduces the research questions. Generation of test cases and mutants are described in Section 5.3.3 and 5.3.4, respectively.

5.3.1 CITO Generation Systems

Figure 5.6 shows the overview of three CITO generation systems. These systems are implemented by us based on the corresponding CITO generation approaches introduced in Section 3.3, because there are no public CITO generation systems. Given a program under the integration testing, CITO generation system takes the information of classes and class dependencies as the input and generates a class test order for integration testing as an output. For each class, class information presents its dependent classes, and the classes on which it depends. Class dependency information indicates attribute coupling and method coupling for each class dependency, which can be used to calculate its stubbing cost. Based on such class and class dependency information, an object relation diagram is constructed for the given program and cycles are identified in the diagram. We assign different weights to each edge measuring the effort of removing it. The appropriate edge is removed to break cycles based on its weight. After all cycles have been broken, a class test order is generated. Three CITO generation systems are implemented based on different kinds of weight calculation:

- NCSys_{tem}: weight is calculated based on the number of cycles. The edge that is involved in the greatest number of cycles will be removed.
- SCSys_{tem}: weight is calculated based on stubbing cost. The edge that requires the lowest stubbing cost will be removed.

- CWRSystem: weight is calculated based on cycles-weight ratio. The edge with the highest cycles-weight ratio is removed.

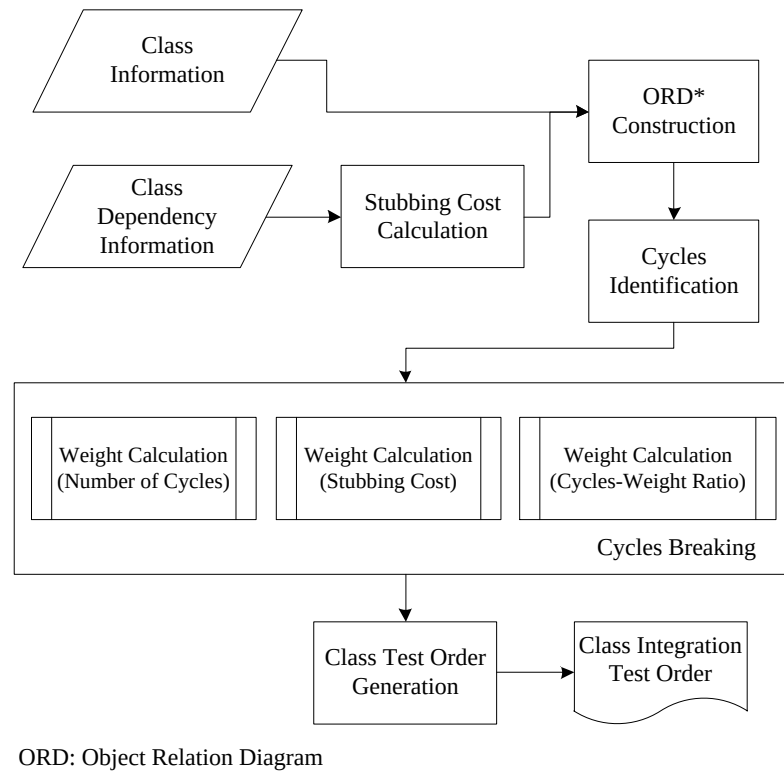


Figure 5.6: Overview of Class Integration Test Order Generation Systems

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

at the same time if all classes on which they depend have been tested. Hence, such multiple classes have the same test order.

5.3.2 Research Questions

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

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

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

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

Because these faults generated by mutation operators are very common in practice. We aim to validate the capability of MT in detecting different kinds of faults. Therefore, we count the number of killed mutants generated by different mutation operators. By comparing the number of different kinds of killed mutants, we can find for which kinds of faults the proposed MRs performed well, and for which kinds of faults the proposed MRs performed the worst, which can provide the direction for our future work, i.e., to propose more specific MRs for those kinds of faults that are difficult to detect now.

5.3.3 Test Case Generation

For the source test cases, we choose four benchmark programs that are popularly used in the CITO generation problem as source inputs. Table 5.1 presents the information of source programs, i.e., these benchmark programs, including the number of classes, class dependencies (Deps), cycles, dependencies that are involved in cycles, and LOC. The detailed information of classes and class dependencies, such as attribute coupling and method coupling, can be found in the literature [11].

Table 5.1: Information of Source Programs

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

We run three CITO generation systems for these benchmark programs and obtain the corresponding class test orders as the source outputs. Follow-up test cases can be generated easily based on source inputs, source outputs, as well as these proposed MRs. For some MRs, more than one follow-up program can be generated according to a source test case, therefore, we randomly choose five follow-up programs in such cases. Table 5.2 counts the number of follow-up programs for each MR. For MR2 and MR3, column "ADD (DEL)" counts the number of follow-up programs generated by adding (deleting) class dependencies for the initial or last class. For MR4, column "SRC (TAR)" counts the number of follow-up programs generated by adding a source (target) class.

As shown in Table 5.2, total 117, 116 and 117 follow-up programs are generated for NCSysytem, SCSSysytem and CWRSSysytem, respectively. For MR1, i.e., the permutation of class name, which is the simplest, five follow-up test cases are generated for each source

test case. For MR2 and MR3, adding class dependencies for the initial (last) class is much easier than deleting class dependencies for the initial (last) class with the aim of preserving its test order. In some cases, such as ANT for NCSys_{tem}, the initial class in the class test order does not depend on any other classes. Similarly, no other classes depend on the last class in the test order. Therefore, no class dependencies from the initial class (or to the last class) can be removed. For MR4, adding a target class on which all the existing classes depend, or a source class which depends on all the existing classes, is easy to implement. Therefore, 20 follow-up test cases are generated for both situations respectively. For MR5, only certain class combinations may preserve the class test order and remain the same stubbing cost. Therefore, it is more complicated to generate such follow-up test cases compared with other MRs.

Table 5.2: Number of Follow-Up Programs for MRs

SUT	Programs	MR1	MR2		MR3		MR4		MR5	SUM
			ADD	DEL	ADD	DEL	SRC	TAR		
NCSys _{tem}	ANT	5	5	0	5	0	5	5	1	26
	ATM	5	5	0	5	1	5	5	2	28
	DNS	5	5	0	5	2	5	5	4	31
	SPM	5	5	3	5	1	5	5	3	32
	SUM	20	20	3	20	4	20	20	10	117
SCSys _{tem}	ANT	5	5	1	5	0	5	5	1	27
	ATM	5	5	0	5	1	5	5	2	28
	DNS	5	5	0	5	2	5	5	4	31
	SPM	5	5	3	5	1	5	5	1	30
	SUM	20	20	4	20	4	20	20	8	116
CWRSys _{tem}	ANT	5	5	1	5	0	5	5	0	26
	ATM	5	5	0	5	1	5	5	2	28
	DNS	5	5	0	5	2	5	5	4	31
	SPM	5	5	3	5	1	5	5	3	32
	SUM	20	20	4	20	4	20	20	9	117

5.3.4 Mutant Generation

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

- **FALSE_RETURNS** (False Returns Mutator): Replace primitive and boxed Boolean return values with false.
- **CONDITIONALS_BOUNDARY** (Conditionals Boundary Mutator): Replace the relational operators, such as, $<$, $>$, with their boundary counterpart, i.e., $<=$, $>=$.
- **EMPTY_RETURNS** (Empty Returns Mutator): Replace return values with an "empty" value for that type. For example, replacing *java.util.List* with *Collections.emptyList()*.
- **INCREMENTS** (Increments Mutator): Replace increments with decrements and vice versa. For example, change increment from 1 to -1 and change increment from -1 to 1.
- **VOID_METHOD_CALLS** (Void Method Call Mutator): Remove method calls to void methods.
- **MATH** (Math Mutator): Replace binary arithmetic operations for integer or floating-point arithmetic with another operation. For example, replace integer subtraction

¹<https://pitest.org/>

with addition, replace double addition with subtraction, replace double division with multiplication, and replace double multiplication with division.

- **NEGATE_CONDITIONALS** (Negate Conditionals Mutator): Replace the original conditional with its negate conditional.
- **NULL_RETURNS** (Null Returns Mutator): Replace return values with null ².
- **PRIMITIVE_RETURNS** (Primitive Returns Mutator): Replace int, short, long, char, float and double return values with 0.

Table 5.3: Information of Mutants for CITO Generation Systems

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

Table 5.3 counts the number of different kinds of mutants generated for three CITO generation systems. Column "NC", "SC" and "CWR" counts the number of different kinds of mutants generated for NCSysyem, SCSysyem and CWRSysyem, respectively. The total number of mutants generated for NCSysyem, SCSysyem, and CWRSysyem is 290, 287, and 286, respectively. More than 30% of mutants are generated by negate conditionals mutator, which replaces the original conditional with the corresponding negate conditional. For

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

example, "==" is replaced by "!=", "<=" is replaced by ">", and ">=" is replaced by "<". Mutants generated by void method call mutator account for 28%, which ranks the second.

Table 5.4 counts the number of mutants generated for cycle-breaking module, which is the most different module among three CITO generation systems. Column "NC", "SC" and "CWR" counts the number of different kinds of mutants generated for NCSys-tem, SCSys-tem and CWRSystem, respectively. The number of mutants generated in cycle-breaking module is 43, 39 and 39 for NCSys-tem, SCSys-tem, and CWRSystem, respectively. Similar with the whole system, almost half of these mutants are generated by negate conditionals mutator.

Table 5.4: Information of Mutants for Cycle-Breaking Module in CITO Generation Systems

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

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

$$MS(P, TS) = \frac{N_k}{N_m - N_e} \quad (5.1)$$

where N_k counts the number of killed mutants by the test suite, N_m counts the total number of mutants and N_e refers to the number of equivalent mutants, which behave the same as the unmutated program. One kind of output information indicates which module our system executed. Therefore, mutants could be regarded as equivalent mutants if the mutated modules were not executed by the system, or the mutants are generated by mutating the statements that print such indication information. Equivalent mutants are excluded when counting the number of mutants in Tables 5.3 and 5.4.

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

5.4 Experimental Results

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

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

For this research question, the generated mutants are used as faulty programs and pairs of test cases are applied in such faulty programs to detect the violation of the proposed MRs. Table 5.5 shows the MS of CITO generation systems for different test suites generated based on five MRs, where all test cases for the same MR are consisted of a test suite. Column N_k counts the number of killed mutants and column MS represents the corresponding mutation scores. More than 84% of faulty programs can be detected by all MRs for three CITO generation systems. Specifically, all 177 pairs of test cases killed 246 mutants and detected 84% of faults for NCSysyem. All pairs of test cases killed 248 mutants and detected 86% of faults for SCSysyem and CWRSysyem. Test cases adopted on SCSysyem and CWRSysyem detected two more faults compared with those applied on NCSysyem. These two faults exist in the function for stubbing cost calculation. While in NCSysyem, stubbing cost is not calculated when breaking cycles because breaking cycles is based on the number of cycles in which each edge involves. Therefore, these two faults are missed in NCSysyem.

Table 5.5: Mutation Scores of CITO Generation Systems for Different Test Suites Generated Based on Five MRs

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

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

While for the simplest MR, the permutation of class names (MR1), its test suite included 20 pairs of test cases, but it only killed 143 mutants for NCSys_{tem} (141 mutants for SCSSys_{tem} and CWRSSys_{tem}). This finding indicates that different MRs own different capability of fault detection, which is consistent with the previous study on MT [76].

More complex MRs are likely to kill more mutants. That is because the follow-up programs generated by more complex MRs, such as MR5, have more different class dependencies from the source programs compared with the follow-up programs generated by simple MRs. For example, Figure 5.7 shows a snippet where a faulty program is generated by changing the conditionals boundary " $i < value$ " to " $i \leq value$ ". The function of this snippet is to store the attribute coupling *value* of a class dependence from the *source* class to the *target* class. For instance, if the attribute coupling *value* is five, then the function *addAttrDeps* will be executed five times to add five different attribute coupling. The faulty program generated by conditionals boundary will count one more time on attribute coupling for all class dependencies. Different values of attribute coupling require different stubbing cost, and correspondingly affect the generated CITO_s. However, such a fault can only be detected by MR5. That is because the follow-up programs generated by MR5 changed the class dependencies to a greater extent compared with other MRs by combining some classes. Specifically, MR1, the permutation of class names, does not affect the class dependencies as well as their attribute coupling. Although MR2 and MR3 add or delete

class dependencies for the initial or last class, for the remaining class dependencies, if we sort them by the stubbing cost, the results of sorting are the same as that in the original programs because all class dependencies are added one more attribute coupling. The situation is the same for MR4. The more different class dependencies in follow-up programs, the more difficult for CITO generation systems to preserve the original results. Therefore, a little modification on the original CITO generation systems may generate different results. In other words, the mutants are easy to be killed.

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

Figure 5.7: A Snippet to Store Attribute Coupling Values for Class Dependencies in CITO Generation Systems

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

is. Because the indexes of the same class are different in source programs and follow-up programs, the generated CITOes will be different, and such a fault can be detected.

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

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

MR4, the performance of test suites generated by adding a target class or a source class is comparable.

Because cycle-breaking module is the most different part among three systems, Table 5.6 presents the MS of cycle-breaking modules for different test suites generated based on five MRs. Column N_k counts the number of killed mutants and column MS represents the corresponding mutation scores. As shown in Table 5.6, more than 65% of faults in cycle-breaking modules can be detected by all MRs. Similar to the whole system, the test suite generated based on MR5 killed the greatest number of mutants for all three systems. The performance of test suites obtained by MR2 and MR3 are comparable, and it is better than the performance of test suites generated by the relatively weaker MRs, such as MR4 and MR1.

Table 5.6: Mutation Scores of Cycle-Breaking Modules for Different Test Suites Generated Based on Five MRs

MRs	Type	NCSys ^{tem}		SCSys ^{tem}		CWRSys ^{tem}	
		N_k	MS	N_k	MS	N_k	MS
MR1	None	22	0.5116	23	0.5897	23	0.5897
MR2	ADD	28	0.6512	28	0.7179	28	0.7179
	DEL	28	0.6512	28	0.7179	28	0.7179
	TOTAL	28	0.6512	28	0.7179	28	0.7179
MR3	ADD	25	0.5814	28	0.7179	28	0.7179
	DEL	26	0.6047	28	0.7179	28	0.7179
	TOTAL	26	0.6047	28	0.7179	28	0.7179
MR4	SRC	20	0.4651	22	0.5641	22	0.5641
	TAR	20	0.4651	22	0.5641	22	0.5641
	TOTAL	20	0.4651	22	0.5641	22	0.5641
MR5	None	28	0.6512	28	0.7179	28	0.7179
Total	None	28	0.6512	28	0.7179	28	0.7179

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

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

Table 5.7 shows MS of three CITO generation systems for different types of mutants. Column N_k counts the number of killed mutants and column MS represents the corresponding mutation scores. To show the results more intuitively, Figures 5.8 to 5.10 present the distribution of these killed mutants for NCSys-tem, SCSys-tem, and CWRSystem, respectively.

Table 5.7: Mutation Scores of CITO Generation Systems for Different Types of Mutants

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

As shown in Table 5.7, except for conditionals boundary mutator and primitive returns mutator, more than 80% of mutants generated by other mutators can be killed, especially for increments mutator, all 16 mutants are killed. Over 90% of faults generated by two mutators (empty returns mutator and void method call mutator) can be detected in NCSys-tem. Similarly, over 90% of faults obtained by two mutators (negate conditionals mutator and null returns mutator) can be detected in SCSys-tem, and over 90% of faults generated by four mutators (void method call mutator, math mutator, negate conditionals mutator and null returns mutator) can be detected in CWRSystem.

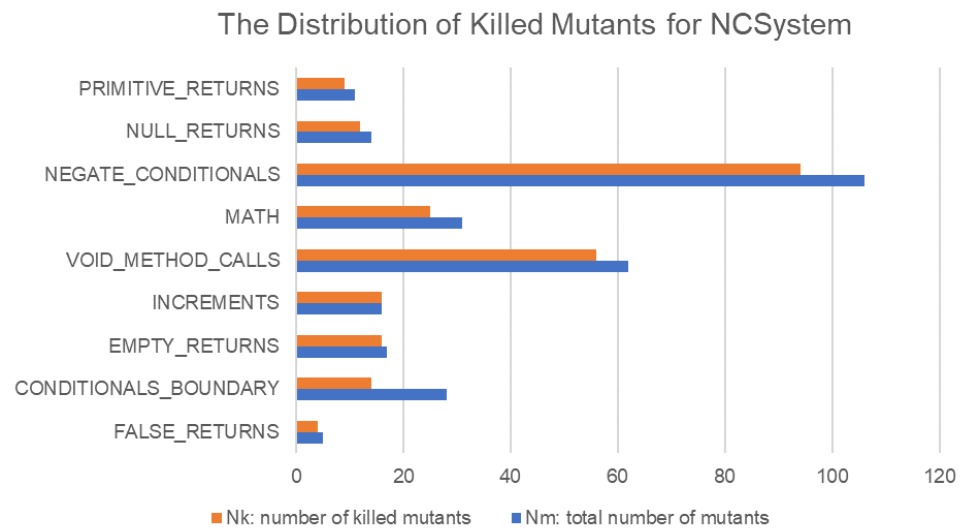


Figure 5.8: The Distribution of Killed Mutants for NCSystem

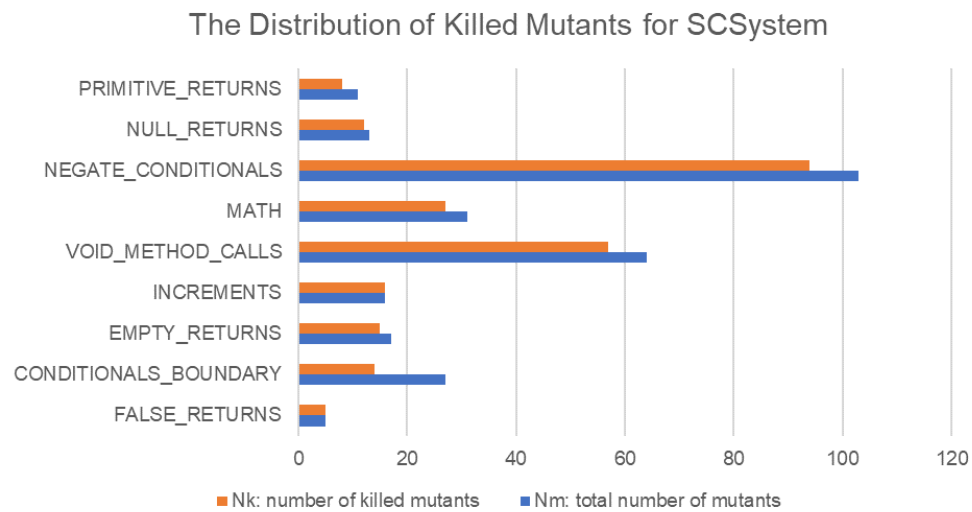


Figure 5.9: The Distribution of Killed Mutants for SCSystem

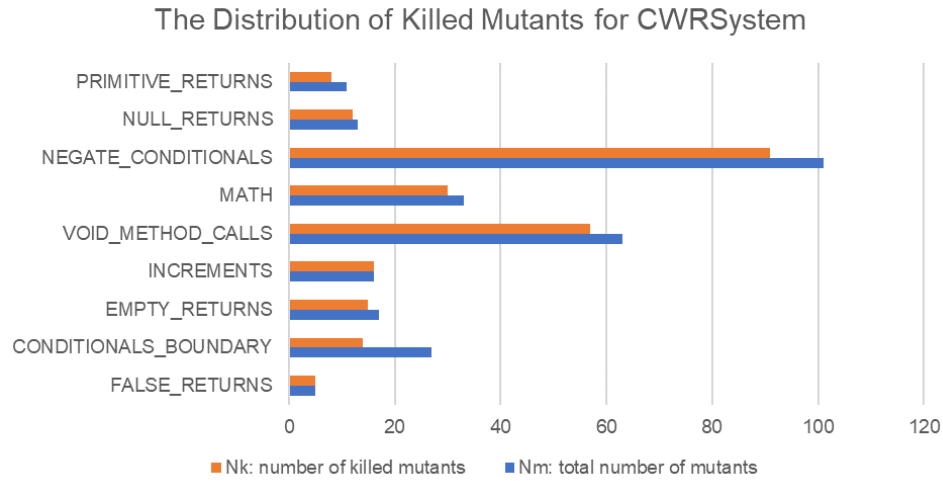


Figure 5.10: The Distribution of Killed Mutants for CWRSystem

Only 50% of mutants generated by conditionals boundary mutator are killed, that is because the relational operator has the same effects on some functions as their boundary counterpart. For example, in function *getMaxStubDep()* which obtains the class dependency with the maximal stubbing cost, the relation operator ">" is used to compare the stubbing cost for two class dependencies and the class dependency with higher stubbing cost will be stored in a temp parameter. In such a case, the effects of ">" and its boundary counterpart ">=" are equal. Similarly, the primitive returns mutator that replaces return values with zero may not violate some proposed MRs. For example, in function *getSCplx()* which calculates stubbing cost for each class dependency, the return value of this function is set to zero by the primitive returns mutator. In such a case, the stubbing cost of any CITOs would be zero, which still conforms to some MRs where the stubbing costs of two class test orders generated by the source program and the follow-up program are expected

to be equal. The currently proposed MRs are unable to detect such faults. Some other criteria should be provided, such as, the stubbing cost of a generated CITO should be greater than zero, which is left for future work.

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

5.5 Discussion

5.5.1 Experimental Results Analysis

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

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

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

Table 5.8 shows the number of cycles for each class dependency in the source program, and its two follow-up programs #1 and #2, respectively. As shown in Table 5.8, for those class dependencies that are already involved in cycles in the source program, most of such class dependencies are involved in more cycles in the follow-up program #1. After adding class dependencies, five more class dependencies are involved in cycles in the follow-up program #1 while only two more class dependencies are involved in cycles in the follow-up program #2. It is obvious that class dependencies in the follow-up program #1 are more

different from the source program. The number of mutants killed by test case #1 and #2 is 216 and 129, respectively. It is consistent with what we analyzed before, that is, the more different class dependencies in follow-up programs, the more difficult for CITO generation systems to preserve the source CITO, and correspondingly, the easier the mutants are killed.

Table 5.8: Number of Cycles for Each Class Dependency in Source Program and Follow-Up Programs for ATM

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

We also investigate whether the generated CITO affects the performance of MT on three CITO generation systems. Table 5.9 shows the generated CITO and OCplx by three systems for the four test suites (ANT, ATM, DNS and SPM). The number in column "CITO" represents the class ID. Table 5.10 shows the number of killed mutants by the four test suites respectively. For the program ANT, the numbers of killed mutants are equal for SCSys and CWRSystem where the generated CITO are the same. Similarly, for the program SPM, the CITO generated by NCSys and CWRSystem are the same, and the numbers of killed mutants are also equal for these two systems. However, for the programs ATM and DNS, although the orders generated by three systems and the corresponding stubbing efforts are equal, the numbers of killed mutants are different, especially for the

program ATM. Therefore, we can conclude that the numbers of killed mutants and the orders generated by three programs are not necessarily related.

Table 5.9: Generated CITOs and OCplx by Three Systems for Four Test Suites

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

Table 5.10: Number of Killed Mutants by Different Test Suites

Test Suites	SUT	N_k
ANT	NCSys _{tem}	234
	SCS _{ystem}	242
	CWRS _{ystem}	242
ATM	NCSys _{tem}	234
	SCS _{ystem}	241
	CWRS _{ystem}	240
DNS	NCSys _{tem}	237
	SCS _{ystem}	233
	CWRS _{ystem}	233
SPM	NCSys _{tem}	245
	SCS _{ystem}	241
	CWRS _{ystem}	245

The more complicated MRs, the stronger the detection capability is. It indicates that more complex MRs that can validate the CITO generation systems may exist. Therefore, except for the five MRs introduced explicitly before, we also present an example for a potentially immature MR, division, which is inspired from MR5 (combining classes with the aim of preserving the original test order). The potentially new MR is dividing some existing classes with the aim of preserving the original test order. Figure 5.11 shows an example of this new MR on NCSys_{tem}. The NCSys_{tem} breaks cycles by removing the class dependency that is involved in the greatest number of cycles. Let SP denote the source program that contains *six* classes, and the class integration test order generated for the program SP is O_{sp} , which is further supposed to be $[C_4, C_5, C_3, C_2, C_1, C_6]$. A follow-up program FP can be generated by dividing the existing class C_4 into two classes (C_4 and C_4') and remaining the existing class dependencies $C_3 \rightarrow C_4$, $C_4 \rightarrow C_1$ and $C_5 \rightarrow C_4$. In the follow-up program, the division of class C_4 does not affect the CITO generated by approaches, therefore, the class integration test order generated for the program FP is similar with O_{sp} ,

that is, $O_{fp} = [C_4, C_4', C_5, C_3, C_2, C_1, C_6]$. We generated test suites based on this sample program to validate the NCSysytem. Although no MRs are violated, test cases generated by six MRs killed 251 mutants for NCSysytem and achieved 0.8655 in mutation score. The test cases generated by the new MR improve the MS by 2%. More complex MRs that can validate the CITO generation systems may exist, which needs our more exploration in the future.

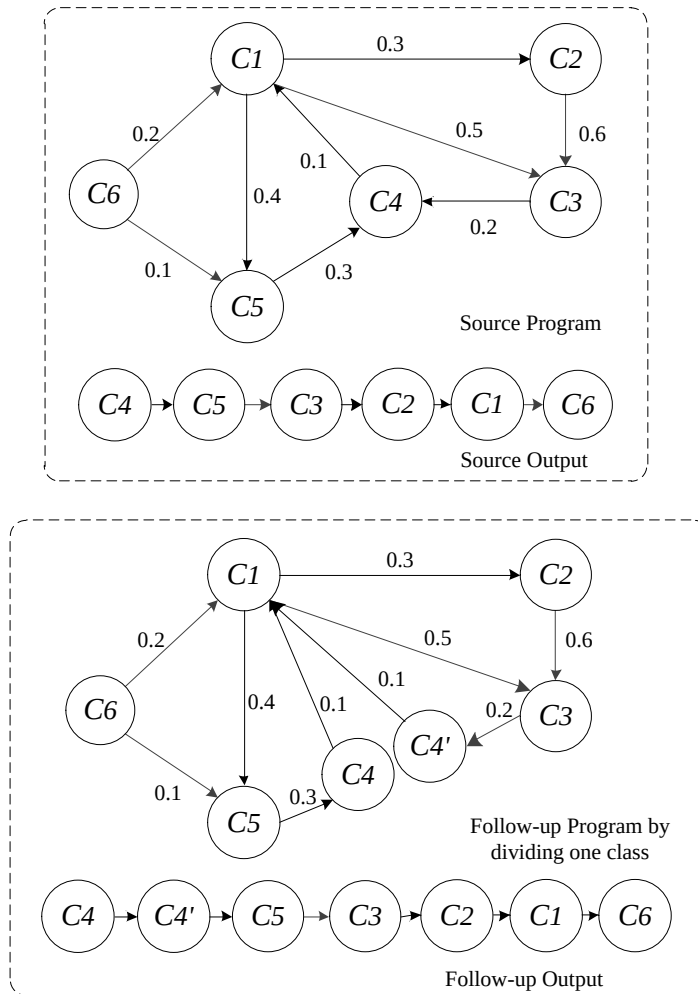


Figure 5.11: Example for New MR (Division) Performed on NCSysytem. Class C_4 is divided into two classes (C_4 and C_4').

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

5.5.2 Threats to Validity

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

For the internal validity, we implemented three CITO generation systems for the experiments because no such systems are public. To check the correctness of our implementation, we generate test suites based on the proposed MRs. Our implemented systems pass all test suites successfully and no MRs are violated.

For the external validity, more than 84% of mutants were killed by around 117 pairs of test cases for three CITO generation systems. As we analyzed in Discussion, even test cases generated by the same MR can own the different capability to kill mutants. To improve the fault detection capability, test cases should be able to generate more different

class dependencies in follow-up programs. We intend to explore how to design such test cases in the future.

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

5.6 Conclusion

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

Chapter 6

Conclusions and Future Work

This thesis has been devoted to the study of class integration test order generation by using several software testing techniques. In this chapter, we summarize our main contributions and discuss some promising future directions.

6.1 Summary

In this thesis, we first proposed a cycle-breaking algorithm ANS in Chapter 3 to improve the effectiveness of the class integration test order generation. It is the first work giving fully consideration between the two factors that affect the overall stubbing cost. The experimental results show that the introduced cycle-breaking algorithm can generate the class integration test orders with lower stubbing cost.

Then, we presented the hypothesis, design, algorithm, and evaluation of CITOSC in Chapter 4, which is a novel system that incorporates similar-class combination into the proposed cycle-breaking algorithm ANS to generate class integration test orders. Specifically, we made the following main contributions:

- A new method is proposed to enhance the efficiency of the class integration test order generation, which is based on the hypothesis that similar-class combination can remove class dependencies and yield a smaller problem space for class integration test order generation. To the best of our knowledge, this is the first work to alleviate the huge search space problem.
- A series of experiments is performed to evaluate the performance of CITOSC. We adopt similar-class combination to the programs under test and compare the reduced programs with the original ones to examine the effects of similar-class combination on minimizing the problem space. We also compare the obtained overall stubbing cost with that of the competing techniques. The empirical results show that CITOSC has the capability to improve the efficiency for the class integration test order generation.

Similar-class combination can help to minimize the search space for programs with a moderately large number of classes. We hope that our study can interest and encourage further research regarding the minimization of the problem space, which can be adopted in not only integration testing, but also other fields of software engineering research that are puzzled by huge search space.

After that, we introduced a validation framework for the class integration test order generation systems using metamorphic testing in Chapter 5. We first proposed several metamorphic relations, which reason about relations between the generated class integration test orders. We then implemented CITOMT, a validation framework for the class integration test order generation systems using these proposed metamorphic relations. The generation systems will be regarded as faulty once a metamorphic relation is violated.

We conducted a series of experiments to evaluate CITOMT, including mutation analysis to assess the effectiveness of different metamorphic relations. The results have successfully

demonstrated the failure detection of CITOMT on the class integration test order generation systems. Moreover, more than 84% of faulty programs are detected by all metamorphic relations.

Our work serves as the first step toward class integration test order generation systems validation. We hope that the promising results will interest and encourage more in-depth research in this new field.

6.2 Future Work

There are several directions for future research. The top priority is to further improve the efficiency and effectiveness for the proposed class integration test order generation approaches.

In this thesis, our hypothesis that similar-class combination can reduce the problem space for large-scale programs that contain a moderately large number of classes is confirmed. Although we have devised two properties to identify similar classes, more characteristics of class dependencies can be analysed to define new kinds of similar classes. The existing two properties, both identical class dependence and symmetric classes are positive associations. Some negative associations also exist among different classes in the program under the test, which needs further exploration. For example, if class *A* inherits from class *B*, then class *A* and *B* should not be tested simultaneously. Therefore, class *A* and class *B* should not be in the search space at the same moment. Based on this rule, the search space can be minimized. Such properties that can describe the negative associations will be investigated in the future.

In addition, we perform a good analysis between two factors that affect the overall stubbing cost, i.e., the number of test stubs and the stubbing cost for each test stub and obtain

some satisfactory results. The analysis of these two factors is only a kind of qualitative analysis in this study, i.e., the fewer the number of test stubs is or the lower the stubbing cost for each test stub is, the lower the overall stubbing cost will be.

As we have known that a conflict between the two factors exists in some cases, further quantitative analysis can be conducted to determine a subtle relationship between these two factors. Also, some criteria can be introduced to measure the impact of two factors on the overall stubbing cost, respectively. Therefore, we can make a better trade-off between the two factors with more information obtained by quantitative analysis. Such measures can further improve the performance of the proposed approach.

In this study, we also validate the class integration test order generation systems based on our proposed metamorphic relations. Such general metamorphic relations can be applied to all class integration test order generation systems, whether they are implemented based on graph-based methods or search-based methods.

As we have known that the fault detection capability of various metamorphic relations is different, more complicated metamorphic relations are required to improve the performance of the proposed validation framework. Because there are no other references to the applicable metamorphic relations, the specific metamorphic relations should be devised for different generation systems respectively, based on the characteristics of different kinds of class integration test order generation approaches. Additionally, there is a need to devise some ideas for automatic generation of metamorphic relations, which can improve the efficiency for validation greatly.

Currently, our study only focuses on Java code. We intend to extend it to other object-oriented programming languages. The main idea of our methods is based on the dependencies among classes, and such dependencies among components including classes are a common phenomenon in object-oriented programs. Therefore, the proposed methods and

validation framework in this study are applicable to other works no matter which kind of programming languages are used.

Bibliography

- [1] Mockito. <http://site.mockito.org>. Online, Accessed 3 Feb, 2016.
- [2] Junit5. <https://junit.org/junit5>. Online, Accessed Sep, 2017.
- [3] A. Abdurazik and J. Offutt. Using coupling-based weights for the class integration and test order problem. *The Computer Journal*, 52:557–570, 2009.
- [4] C. Aruna and R. S. R. Prasad. Testing approach for dynamic web applications based on automated test strategies. In *ICT and Critical Infrastructure: Proceedings of the 48th Annual Convention of Computer Society of India- Vol II*, pages 399–410, 2014.
- [5] W. K. G. Assunção, T. E. Colanzi, S. R. Vergilio, and A. Pozo. A multi-objective optimization approach for the integration and test order problem. *Information Sciences*, 267:119–139, 2014.
- [6] P. Bansal, S. Sabharwal, and 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)*, pages 1–8, 2009.

- [7] E. T. Barr, M. Harman, P. McMinn, M. Shahbaz, and S. Yoo. The oracle problem in software testing: a survey. *IEEE Transactions on Software Engineering*, 41(5):507–525, 2015.
- [8] B. Beizer. Software testing techniques. *Van Nostrand Reinhold, New York, 2nd Ed.*, 1990.
- [9] S. Beydeda. Self-metamorphic-testing components. In *Proceedings of the 30th Annual International Computer Software and Applications Conference (COMP-SAC’06)*, volume 2, pages 265–272, 2006.
- [10] L. Borner and 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*, pages 80–85, 2009.
- [11] L. C. Briand, J. Feng, and Y. Labiche. Experimenting with genetic algorithms to devise optimal integration test orders. *Tech. Rep. TR SCE-02-03, Carleton University, Springer US*, pages 204–234, 2003.
- [12] L. C. Briand, Y. Labiche, and Y. Wang. An investigation of graph-based class integration test order strategies. *IEEE Transactions on Software Engineering*, 29(7):594–607, 2003.
- [13] R. E. Burkard and F. Rendl. A thermodynamically motivated simulation procedure for combinatorial optimization problems. *European Journal of Operational Research*, 17(2):169 – 174, 1984.
- [14] E. K. Burke, M. R. Hyde, and G. Kendall. Grammatical evolution of local search heuristics. *IEEE Transactions on Evolutionary Computation*, 16(3):406–417, 2011.

- [15] E. K. Burke, M. R. Hyde, G. Kendall, G. Ochoa, E. Ozcan, and 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.
- [16] C. Cadar and K. Sen. Symbolic execution for software testing: Three decades later. *Commun. ACM*, 56(2):82–90, 2013.
- [17] P. C. Cañizares, A. Núñez, M. Núñez, and J. J. Pardo. A methodology for designing energy-aware systems for computational science. *Procedia Computer Science*, 51:2804–2808, 2015.
- [18] Y. Cao, Z. Q. Zhou, and T. Y. Chen. On the correlation between the effectiveness of metamorphic relations and dissimilarities of test case executions. In *Proceedings of 2013 13th International Conference on Quality Software*, pages 153–162, 2013.
- [19] C. Castro-Cabrera and I. Medina-Bulo. An approach to metamorphic testing for WS-BPEL compositions. In *Proceedings of the International Conference on e-Business*, pages 1–6, 2011.
- [20] C. Castro-Cabrera and I. Medina-Bulo. Application of metamorphic testing to a case study in web services compositions. *Communications in Computer and Information Science*, 314:168–181, 2012.
- [21] F. T. Chan, T. Y. Chen, S. C. Cheung, M. F. Lau, and S. M. Yiu. Application of metamorphic testing in numerical analysis. In *Proceedings of the IASTED International Conference on Software Engineering*, pages 191–197, 1998.

- [22] W. K. Chan, T. Y. Chen, H. Lu, T. H. Tse, and S. S. Yau. Integration testing of context-sensitive middleware-based applications: A metamorphic approach. *International Journal of Software Engineering and Knowledge Engineering*, 16(05):677–703, 2006.
- [23] W. K. Chan, S. C. Cheung, and K. R.P.H. Leung. A metamorphic testing approach for online testing of service-oriented software applications. *International Journal of Web Services Research*, 4(2):60–80, 2007.
- [24] W. K. Chan, J. C. F. Ho, and T. H. Tse. Piping classification to metamorphic testing: An empirical study towards better effectiveness for the identification of failures in mesh simplification programs. In *Proceedings of the 31th Annual International Computer Software and Applications Conference (COMPSAC 2007)*, pages 397–404, 2007.
- [25] W. K. Chan, J. C. F. Ho, and T. H. Tse. Finding failures from passed test cases: Improving the pattern classification approach to the testing of mesh simplification programs. *Software Testing, Verification and Reliability*, 20(2):89–120, 2010.
- [26] T. Y. Chen, S. C. Cheung, and S. M. Yiu. Metamorphic testing: A new approach for generating next test cases. *Tech. Rep. HKUST-CS98-01, Department of Computer Science, The Hong Kong University of Science and Technology*, 1998.
- [27] T. Y. Chen, J. W. K. Ho, H. Liu, and X. Xie. An innovative approach for testing bioinformatics programs using metamorphic testing. *BMC Bioinformatics*, 10:1 – 12, 2009.

- [28] T. Y. Chen, D. H. Huang, T.H. Tse, and Z. Q. Zhou. Case studies on the selection of useful relations in metamorphic testing. In *Proceedings of the 4th Ibero-American Symposium on Software Engineering and Knowledge Engineering (JISIC'04)*, pages 569–583, 2004.
- [29] T. Y. Chen, F. Kuo, W. Ma, W. Susilo, D. Towey, J. Voas, and Z. Q. Zhou. Metamorphic testing for cybersecurity. *Computer*, 49(6):48–55, 2016.
- [30] T. Y. Chen, F.-C. Kuo, H. Liu, P.-L. Poon, D. Towey, T. H. Tse, and Z. Q. Zhou. Metamorphic testing: A review of challenges and opportunities. *ACM Comput. Surv.*, 51(1):1–27, 2018.
- [31] T. Y. Chen, P.-L. Poon, and X. Xie. Metric: Metamorphic relation identification based on the category-choice framework. *Journal of Systems and Software*, 116:177–190, 2016.
- [32] T. Y. Chen, T. H. Tse, and Z. Q. Zhou. Semi-proving: An integrated method for program proving, testing, and debugging. *IEEE Transactions on Software Engineering*, 37(1):109–125, 2011.
- [33] T. Y. Chen, T.H. Tse, and Z. Q. Zhou. Semi-proving:an integratedmethod based on global symbolic evaluation and metamorphic testing. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, (ASE'18), pages 132–142, 2018.
- [34] G. Czibula, I. G. Czibula, and Z. Marian. An effective approach for determining the class integration test order using reinforcement learning. *Appl. Soft Comput.*, 65(C):517–530, 2018.

- [35] R. da Veiga Cabral, A. Pozo, and 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*, pages 16–29, Berlin, Heidelberg, 2010.
- [36] K. Deb, S. Agrawal, A. Pratap, and 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)*, pages 849–858, Berlin, Heidelberg, 2000.
- [37] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multi-objective genetic algorithm: NSGA-II. *IEEE Transactions on Evolutionary Computation*, 6(2):182–197, 2002.
- [38] J. L. Devore. Probability and statistics for engineering and the sciences. *5th ed.* Duxbury Press, 1999.
- [39] J. Ding, X. Hu, and V. Gudivada. A machine learning based framework for verification and validation of massive scale image data. *IEEE Trans. on Big Data*, pages 1–1, 2017.
- [40] J. Ding, T. Wu, D. Wu, J. Q. Lu, and X. Hu. Metamorphic testing of a monte carlo modeling program. In *Proceedings of the 6th International Workshop on Automation of Software Test, AST '11*, pages 1–7. Association for Computing Machinery, 2011.
- [41] K. Doerner, W. J. Gutjahr, R. F. Hartl, C. Strauss, and C. Stummer. Pareto ant colony optimization: A metaheuristic approach to multiobjective portfolio selection. *Annals of Operations Research*, 131(1):79–99, 2004.

- [42] G. Dong, B. Xu, L. Chen, C. Nie, and L. Wang. Survey of metamorphic testing. *Journal of Frontiers of Computer Science and Technology*, 3(2):130–143, 2009.
- [43] M. Dorigo and K. Socha. An introduction to ant colony optimization. *Computer Science Technical Report No. TR/IRIDIA/2006-010, Universit de Libre de Bruxelles, CP*, 194(6), 2006.
- [44] A. Dwarakanath, M. Ahuja, S. Sikand, R. M. Rao, R. P. J. C. Bose, N. Dubash, and S. Podder. Identifying implementation bugs in machine learning based image classifiers using metamorphic testing. In *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, (ISSTA’18), pages 118–128, New York, NY, USA, 2018.
- [45] J. E. Gewehr, M. Szugat, and R. Zimmer. BioWeka-extending the Weka framework for bioinformatics. *Bioinformatics*, 23(5):651–653, 2007.
- [46] P. Godefroid, N. Klarlund, and K. Sen. Dart: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI’05, pages 213–223, New York, NY, USA, 2005.
- [47] M. Grechanik and G. Devanla. Generating integration tests automatically using frequent patterns of method execution sequences. In *SEKE*, pages 209–280, 2019.
- [48] R. Guderlei and J. Mayer. Towards automatic testing of imaging software by means of random and metamorphic testing. *International Journal of Software Engineering and Knowledge Engineering*, 17(06):757–781, 2007.
- [49] G. Guizzo, M. Bazargani, M. Paixao, and 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)*, pages 168–174, Cham, 2017.
- [50] G. Guizzo, G. M. Fritsche, S. R. Vergilio, and 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*, (GECCO’15), pages 1343–1350, New York, NY, USA, 2015.
- [51] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.
- [52] V. L. Hanh, K. Akif, Y. Le Traon, and 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*, pages 381–401, Berlin, Heidelberg, 2001.
- [53] M. Harman, E. K. Burke, J. Clark, and 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)*, pages 1–8, New York, NY, USA, 2012.
- [54] M. Harman, Y. Jia, and Y. Zhang. Achievements, open problems and challenges for search based software testing. In *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, pages 1–12, 2015.
- [55] N. L. Hashim, H. W. Schmidt, and S. Ramakrishnan. Test order for class-based integration testing of Java applications. In *Proceedings of the 5th International Conference on Quality Software (QSIC’05)*, pages 11–18, 2005.

- [56] T. Hastie, R. Tibshirani, and J. Friedman. *The Elements of Statistical Learning*, Springer, 2nd Ed., 2009.
- [57] R. Hewett and 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)*, pages 211–220, Nov 2009.
- [58] P. Hu, Z. Zhang, W. K. Chan, and T. H. Tse. An empirical comparison between direct and indirect test result checking approaches. In *Proceedings of the 3rd International Workshop on Software Quality Assurance, SOQUA'06*, pages 6–13, New York, NY, USA, 2006.
- [59] Z.-W. Hui and S. Huang. Achievements and challenges of metamorphic testing. In *Proceedings of the 2013 Fourth World Congress on Software Engineering*, pages 73–77, 2013.
- [60] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and control-flow-based test adequacy criteria. In *Proceedings of 16th International Conference on Software Engineering*, pages 191–200, 1994.
- [61] M. Jiang, T. Y. Chen, F.-C. Kuo, and Z. Ding. Testing central processing unit scheduling algorithms using metamorphic testing. In *2013 IEEE 4th International Conference on Software Engineering and Service Science*, pages 530–536, 2013.
- [62] M. Jiang, T. Y. Chen, F.-C. Kuo, Z. Ding, E. Choi, and O. Mizuno. A revisit of the integration of metamorphic testing and test suite based automated program repair. In *Proceedings of the 2nd International Workshop on Metamorphic Testing (MET'17)*, pages 14–20, 2017.

- [63] M. Jiang, T. Y. Chen, F.-C. Kuo, D. Towey, and Z. Ding. A metamorphic testing approach for supporting program repair without the need for a test oracle. *Journal of Systems and Software*, 126:127–140, 2017.
- [64] H. Jin, Y. Jiang, N. Liu, C. Xu, X. Ma, and J. Lu. Concolic metamorphic debugging. In *2015 IEEE 39th Annual Computer Software and Applications Conference*, volume 2, pages 232–241, 2015.
- [65] R. Just and F. Schweiggert. Evaluating testing strategies for imaging software by means of mutation analysis. In *2009 International Conference on Software Testing, Verification, and Validation Workshops*, pages 205–209, 2009.
- [66] U. Kanewala, J. M. Bieman, and A. Ben-Hur. Predicting metamorphic relations for testing scientific software: A machine learning approach using graph kernels. *Softw. Test. Verif. Reliab.*, 26(3):245–269, May 2016.
- [67] D. C. Kung, J. Gao, P. Hsia, J. Lin, and Y. Toyoshima. Class firewall, test order, and regression testing of Object-Oriented programs. *J. Object-Oriented. Program*, 8:51–65, 05 1995.
- [68] D. C. Kung, J. Gao, P. Hsia, Y. Toyoshima, and C. Chen. On regression testing of Object-Oriented programs. *Journal of Systems and Software*, 32(1):21–40, 1996.
- [69] F.-C. Kuo, T. Y. Chen, and W. K. Tam. Testing embedded software by metamorphic testing: A wireless metering system case study. In *2011 IEEE 36th Conference on Local Computer Networks*, pages 291–294, 2011.
- [70] F.-C. Kuo, S. Liu, and T. Y. Chen. Testing a binary space partitioning algorithm with metamorphic testing. In *Proceedings of the 2011 ACM Symposium on Applied Computing, SAC’11*, pages 1482–1489, New York, NY, USA, 2011.

- [71] V. Le, M. Afshari, and Z. Su. Compiler validation via equivalence modulo inputs. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, pages 216–226, 2014.
- [72] Y. Le Traon, T. Jérón, J. Jézéquel, and P. Morel. Efficient Object-Oriented integration and regression testing. *IEEE Transactions on Reliability*, 49(1):12–25, 2000.
- [73] C. Lidbury, A. Lascu, N. Chong, and A. F. Donaldson. Many-core compiler fuzzing. *SIGPLAN Not.*, 50(6):65–76, 2015.
- [74] L. J. Lin. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning*, 8(3-4):293–321, 1992.
- [75] M. Lindvall, D. Ganesan, R. Árdal, and R. E. Wiegand. Metamorphic model-based testing applied on NASA DAT: An experience report. In *Proceedings of the 37th International Conference on Software Engineering (ICSE'15)*, pages 129–138, 2015.
- [76] H. Liu, F. Kuo, D. Towey, and T. Y. Chen. How effectively does metamorphic testing alleviate the oracle problem? *IEEE Trans. Software Eng*, 40(1):4–22, 2014.
- [77] H. Liu, X. Liu, and T. Y. Chen. A new method for constructing metamorphic relations. In *2012 12th International Conference on Quality Software*, pages 59–68, 2012.
- [78] H. Liu, I. I. Yusuf, H. W. Schmidt, and T. Y. Chen. Metamorphic fault tolerance: An automated and systematic methodology for fault tolerance in the absence of test oracle. In *Companion Proceedings of the 36th International Conference on Software Engineering, ICSE Companion 2014*, pages 420–423, New York, NY, USA, 2014.

- [79] E. L. Lloyd, M. L. Soffa, and C.-C. Wang. On locating minimum feedback vertex sets. *Journal of Computer and System Sciences*, 37:292–311, 1988.
- [80] X. Lu, Y. Dong, and C. Luo. Testing of component-based software: A metamorphic testing methodology. In *2010 7th International Conference on Ubiquitous Intelligence Computing and 7th International Conference on Autonomic Trusted Computing*, pages 272–276, 2010.
- [81] B. A. Malloy, P. J. Clarke, and 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)*, pages 353–364, 2003.
- [82] L. I. Manolache and D. G. Kourie. Software testing using model programs. *Softw. Pract. Exp.*, 31(13):1211–1236, 2001.
- [83] T. Mariani, G. Guizzo, S. R. Vergilio, and 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*, (GECCO’16), pages 1069–1076, New York, NY, USA, 2016.
- [84] J. Mayer and R. Guderlei. An empirical study on the selection of good metamorphic relations. In *30th Annual International Computer Software and Applications Conference (COMPSAC’06)*, volume 1, pages 475–484, 2006.
- [85] J. Mayer and R. Guderlei. On random testing of image processing applications. In *2006 Sixth International Conference on Quality Software (QSIC’06)*, pages 85–92, 2006.

- [86] H. Melton and E. Tempero. An empirical study of cycles among classes in Java. *Empirical Software Engineering*, 12(4):389–415, 2007.
- [87] N. Mouha, M. S. Raunak, D. R. Kuhn, and R. Kacker. Finding bugs in cryptographic hash function implementations. *IEEE Trans. on Relia*, 67(3):870–884, 2018.
- [88] C. Murphy, G. Kaiser, L. Hu, and L. Wu. Properties of machine learning applications for use in metamorphic testing. *Tech. Rep. CUCS-011-08, Dept. Comput. Sci., Columbia Univ., New York NY, USA*, 2008.
- [89] C. Murphy, G. Kaiser, L. Hu, and L. Wu. Properties of machine learning applications for use in metamorphic testing. In *Proceedings of International Conference on Software Engineering & Knowledge Engineering*, pages 867–872, 2008.
- [90] S. Nakajima and H. N. Bui. Dataset coverage for testing machine learning computer programs. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC’16)*, pages 297–304, 2016.
- [91] A. Núñez and R. Hierons. A methodology for validating cloud models using metamorphic testing. *annals of telecommunications - annales des télécommunications*, 70:127–135, 2014.
- [92] R.A.P. Oliveira, U. Kanewala, and P. A. Nardi. Automated test oracles: State of the art, taxonomies, and trends. *Advances in Computers*, 95:113–199, 2014.
- [93] M. Olsen and M. Raunak. Increasing validity of simulation models through metamorphic testing. *IEEE Transactions on Reliability*, 68(1):91–108, 2019.
- [94] T. Orenstein, Z. Kohavi, and I. Pomeranz. An optimal algorithm for cycle breaking in directed graphs. *Journal of Electronic Testing*, 7.1-2(4):71–81, 1995.

- [95] V. Pareto. *Manuel D'Économie Politique*. Ams Press, Paris, 1927.
- [96] K. Patel and R. M. Hierons. A mapping study on testing non-testable systems. *Software Quality Journal*, 26(4):1373–1413, 2018.
- [97] D. Pesu, Z. Q. Zhou, J. Zhen, and D. Towey. A monte carlo method for metamorphic testing of machine translation services. In *Proceedings of the 3rd International Workshop on Metamorphic Testing*, (MET'18), pages 38–45, New York, NY, USA, 2018.
- [98] M. Pezzè and C. Zhang. Automated test oracles: A survey. *Advances in Computers*, 95:1–48, 2014.
- [99] S. Pugh, M. S. Raunak, D. R. Kuhn, and R. Kacker. Systematic testing of post-quantum cryptographic implementations using metamorphic testing. In *Proceedings of the 4th International Workshop on Metamorphic Testing (MET'19)*, pages 2–8, 2019.
- [100] L. L. Pullum and O. Ozmen. Early results from metamorphic testing of epidemiological models. In *Proceedings of the 2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom'12)*, pages 62–67, 2012.
- [101] A. Ramanathan, C. A. Steed, and L. L. Pullum. Verification of compartmental epidemiological models using metamorphic testing, model checking and visual analytics. In *Proceedings of the 2012 ASE/IEEE International Conference on BioMedical Computing (BioMedCom'12)*, pages 68–73, 2012.
- [102] P. Rao, Z. Zheng, T. Y. Chen, N. Wang, and K. Cai. Impacts of test suite's class imbalance on spectrum-based fault localization techniques. In *2013 13th International Conference on Quality Software*, pages 260–267, 2013.

- [103] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Trans. Softw. Eng.*, 21(1):19–31, 1995.
- [104] C. Ryan, J. J. Collins, and M. O. Neill. Grammatical evolution: Evolving programs for an arbitrary language. In *Proceedings of the 1998 European Conference on Genetic Programming*, pages 83–96, 1998.
- [105] M. S. Sadi, F.-C. Kuo, J. W. K. Ho, M. A. Charleston, and T. Y. Chen. Verification of phylogenetic inference programs using metamorphic testing. *Journal of Bioinformatics and Computational Biology*, 09(06):729–747, 2011.
- [106] S. Segura, A. Durán, A. B. Sánchez, D. L. Berre, E. I. Lonca, and A. Ruiz-Cortés. Automated metamorphic testing of variability analysis tools. *Software Testing, Verification and Reliability*, 25(2):138–163, 2015.
- [107] S. Segura, G. Fraser, A. B. Sánchez, and A. Ruiz-Cortés. A survey on metamorphic testing. *IEEE Trans. Software Eng.*, 42(9):805–824, 2016.
- [108] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated test data generation on the analyses of feature models: A metamorphic testing approach. In *2010 Third International Conference on Software Testing, Verification and Validation*, pages 35–44, 2010.
- [109] S. Segura, R. M. Hierons, D. Benavides, and A. Ruiz-Cortés. Automated metamorphic testing on the analyses of feature models. *Information and Software Technology*, 53(3):245–258, 2011.
- [110] S. Segura, J. A. Parejo, J. Troya, and A. Ruiz-Cortés. Metamorphic testing of RESTful web APIs. *IEEE Transactions on Software Engineering*, 44(11):1083–1099, 2018.

- [111] S. Segura, J. Troya, A. Durán, and A. Ruiz-Cortés. Performance metamorphic testing: A proof of concept. *Information and Software Technology*, 98:1–4, 2018.
- [112] K. Sen, D. Marinov, and G. Agha. CUTE: A concolic unit testing engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 263–272, New York, NY, USA, 2005.
- [113] M. Steindl and J. Mottok. Optimizing software integration by considering integration test complexity and test effort. In *Proceedings of the 10th International Workshop on Intelligent Solutions in Embedded Systems*, pages 63–68, 2012.
- [114] C. Sun, G. Wang, B. Mu, H. Liu, Z. Wang, and T.Y. Chen. Metamorphic testing for web services: Framework and a case study. In *2011 IEEE International Conference on Web Services*, pages 283–290, 2011.
- [115] R. S. Sutton and A. G. Barto. *Reinforcement learning: An introduction*. Cambridge, MA: MIT Press, 2011.
- [116] S. Tahvili. *Multi-Criteria Optimization of System Integration Testing*. Mälardalen University College, Västerås, Eskilstuna, Sweden, GRIN Verlag, 2019.
- [117] K. C. Tai and 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)*, pages 602–607, 1997.
- [118] Q. Tao, W. Wu, C. Zhao, and W. Shen. An automatic testing approach for compiler based on metamorphic testing technique. In *Proceedings of the 2010 Asia Pacific Software Engineering Conference (APSEC’10)*, pages 270–279, 2010.

- [119] R. Tarjan. Depth-first search and linear graph algorithms. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (SWAT 1971)*, pages 114–121, 1971.
- [120] Q. T. Tran. Java applet CPU scheduling algorithm animations. <http://www.utdallas.edu/~ilyen/animation/Icpu/Iprogram/Iprog.html>, 1998.
- [121] J. Troya, S. Segura, and A. Ruiz-Cortés. Automated inference of likely metamorphic relations for model transformations. *Journal of Systems and Software*, 136:188–208, 2018.
- [122] M. Črepinšek, S.-H. Liu, and M. Mernik. Exploration and exploitation in evolutionary algorithms: A survey. *ACM Comput. Surv.*, 45(3):1–33, July 2013.
- [123] S. R. Vergilio, A. Pozo, J. C. G. Árias, R. da Veiga Cabral, and T. Nobre. Multi-objective optimization algorithms applied to the class integration and test order problem. *International Journal on Software Tools for Technology Transfer*, 14:461–475, 2012.
- [124] Y. Wang, Z. Zhu, H. Yu, and B. Yang. Risk analysis on multi-granular flow network for software integration testing. *IEEE Transactions on Circuits and Systems II: Express Briefs*, 65(8):1059–1063, 2018.
- [125] X. Xie, J. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen. Application of metamorphic testing to supervised classifiers. In *2009 Ninth International Conference on Quality Software*, pages 135–144, 2009.
- [126] X. Xie, J. W.K. Ho, C. Murphy, G. Kaiser, B. Xu, and T. Y. Chen. Testing and validating machine learning classifiers by metamorphic testing. *Journal of Systems and Software*, 84(4):544–558, 2011.

- [127] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu. Spectrum-based fault localization: Testing oracles are no longer mandatory. In *2011 11th International Conference on Quality Software*, pages 1–10, 2011.
- [128] X. Xie, W. E. Wong, T. Y. Chen, and B. Xu. Metamorphic slice: An application in spectrum-based fault localization. *Inf. Softw. Technol.*, 55(5):866–879, 2013.
- [129] X. Xie, Z. Zhang, T. Y. Chen, Y. Liu, P.-L. Poon, and B. Xu. METTLE: A METamorphic Testing approach to assessing and validating unsupervised machine LEarning systems. *IEEE Transactions on Reliability*, 69(4):1293–1322, 2020.
- [130] L. Xu, D. Towey, A. P. French, S. Benford, Z. Q. Zhou, and T. Y. Chen. Enhancing supervised classifications with metamorphic relations. In *Proceedings of the 3rd International Workshop on Metamorphic Testing*, (MET’18), pages 46–53, New York, NY, USA, 2018.
- [131] M. Zhang, Y. Zhang, L. Zhang, C. Liu, and S. Khurshid. DeepRoad: GAN-based metamorphic testing and input validation framework for autonomous driving systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, (ASE’18), pages 132–142, New York, NY, USA, 2018.
- [132] Z. Zhang, W. K. Chan, T. H. Tse, and P. Hu. Experimental study to compare the use of metamorphic testing and assertion checking. *Journal of Software*, 20(10):2637–2654, 2009.
- [133] Z. Q. Zhou and L. Sun. Metamorphic testing of driverless cars. *Commun. ACM*, 62(3):61–67, 2019.

- [134] Z. Q. Zhou, L. Sun, T. Y. Chen, and D. Towey. Metamorphic relations for enhancing system understanding and use. *IEEE Transactions on Software Engineering*, 46(10):1120–1154, 2018.
- [135] Z. Q. Zhou, T. H. Tse, F.-C. Kuo, and T. Y. Chen. Automated functional testing of web search engines in the absence of an oracle. *Technical Report TR-2007-06, Department of Computer Science, The University of Hong Kong*, 2007.
- [136] Z. Q. Zhou, S. Xiang, and T. Y. Chen. Metamorphic testing for software quality assessment: A study of search engines. *IEEE Transactions on Software Engineering*, 42(3):264–284, 2016.
- [137] Z. Q. Zhou, S. Zhang, M. Hagenbuchner, T. H. Tse, Fei-Ching Kuo, and T. Y. Chen. Automated functional testing of online search services. *Software Testing, Verification and Reliability*, 22(4):221–243, 2012.
- [138] H. Zhu. JFuzz: A tool for automated Java unit testing based on data mutation and metamorphic testing methods. In *2015 Second International Conference on Trustworthy Systems and Their Applications*, pages 8–15, 2015.
- [139] H. Zhu, P. A. V. Hall, and J. H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.