博士論文

Automatic Verification and Testing for Software with Multiple Versions
-- Improving Software Quality in the Era of Multiple Versions --
(複数バージョンのあるソフトウェアの自動検証・検査 -- 複数バージョン管理時代のソフトウェアの品質向上 --)

馬　雷
Lei Ma

Automatic Verification and Testing for Software with Multiple Versions

-Improving Software Quality in the Era of Multiple Versions-

Lei Ma

A PHD Thesis

in

Electrical Engineering and Information System

Submitted to the Graduate School of Engineering
the University of Tokyo

in

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

June, 2014

Hiroyuki Sato
Supervisor of Thesis

ABSTRACT

The verification and testing of software systems to improve the software quality are two major significant activities in the software development cycle. However, while their significance is widely known, it is also known that occupy large part of cost in software development and maintenance. According to Lehman's software evolution laws, a software system has to be continuously changed to increase its functionalities, to fix bugs, to and adapt for new requirements over its lifecycle. Such changes are released as a series of updated software versions that share many commonalities among multiple versions. As the widely adoption of revision control system, Software Product Lines, and *clone-and-own* techniques, more and more similar version variants are produced. This brings new challenges for conventional verification and testing techniques, which are only able to analyze one single version.

Separate verification and testing for each individual version would cause many redundancies in analyzing the same code. In other words, the results from separate analysis processes are also difficult to be shared among multiple versions. Sharing common knowledge is important to improve the overall analysis results and achieve better quality assurance guarantee such as code coverage. Therefore, there is a strong demand to create and design novel verification and testing techniques for multiple versions, improving the overall quality guarantee for all versions efficiently. However, the simultaneous analysis of multiple versions is inherently challenging, especially for managed languages like Java and C#, and platforms designed to handle a single version.

In this thesis, we first propose a general concept and framework *project centralization* to manage multiple versions. Project centralization shares commonalities of multiple versions as much as possible while preserving the behaviors (semantics) of each version. We formalize the version conflict problem and propose a graph representation for all versions of products under analysis. Based on this representation, we transform the project centralization problem into a graph coloring problem where existing solutions can be applied to calculate both the optimal and near optimal solutions. We implement different existing techniques and compare their effectiveness for project centralization. We can conclude that our proposed heuristic algorithm is both efficient and effective to calculate the near optimal solution. Based on this framework, we perform consecutive of studies, implement many

software verification and testing tool chains, and show that the our tools are useful for improving software quality and correctness of general multiple version software.

Distributed systems are typical complicated software examples that are operated with multiple versions. By extending project centralization, we further propose *process centralization* techniques for such challenging software quality insurance fields, where large combinational states and interactive network communications between peers, and concurrency are involved. With combined project centralization and process centralization approaches, we have successfully applied our tools to verify some practical distributed applications with multiple versions, demonstrating the effectiveness of our technique in revealing bugs that are unable to be detected by using existing techniques.

In additional to verification, testing is also a major approach for detecting software defect and improving software quality. Among various testing techniques, automatic testing techniques like random testing are proved to be useful in finding bugs and improving software quality. Conventional techniques allow to test only one single product. We further propose novel program analysis enhanced testing approaches and centralization based approaches for multiple versions.

We propose fully automatic enhanced automatic testing techniques by adopting domain knowledge of software under testing. Our technique automatically performs program analysis on the software under test. Combined with runtime feedback during testing, our technique uses both static and dynamic analysis to guide testing. Our design and implementation outperform the current most advanced fully automatic random testing tool Randoop after many strict and thorough evaluations on more than 30 widely used benchmarks and by researchers on their developed collection products inside Software Competence Center Hagenberg (SCCH).

To perform automatic testing for multiple versions, we further refine our project centralization techniques and design novel testing strategies for multiple versions. Our techniques on real-world benchmarks demonstrate that the reuse of analysis results can improve the overall performance.

In summary, this thesis focuses on the issues to improve software quality, specifically on proposing novel techniques for the management, verification and testing multiple versions (with version conflicts) in the era of many coexisted version variants. We summarize the main achievements of our proposed concepts, techniques and implementations from our consecutive studies on the management, verification

and testing to improve software quality for multiple versions. Our work successfully makes steps further and solves important problems for multiple version related analysis: (1) manage multiple version variants, sharing common code while preserve the behavior of each version. (2) The verification of distributed application, with multiple version coexisted and running on multiple peers. (3) Improving automatic testing techniques and design novel testing approach for multiple versions, although it may just represent a small piece of the whole iceberg: the verification and testing research for multiple versions, where many other problems are still needed to be solved. As more and more problems are caused by the multiple versions recently, more novel fundamental and practical techniques and solutions are needed from research community, which is expected to be one of the most important issues and research topics in software quality insurance in the near future.

Contents

1 Introduction **1**
1.1 Background and Motivation . . . 1
1.2 Version Conflict Issues and Project Centralization . . . 4
1.3 Distributed System and the Challenge in its Verification . . . 8
1.4 Software Testing for Multiple Version Variants . . . 11
1.4.1 Variants of Random Testing . . . 12
1.4.2 Use of Domain Knowledge . . . 13
1.4.3 Symbolic Execution . . . 14
1.4.4 Automatic Testing Techniques for Multiple Versions . . . 14
1.5 Thesis Outline . . . 15
1.6 Research Contributions . . . 16

2 Project Centralization **18**
2.1 Concepts in Project Centralization . . . 18
2.1.1 A Worklist Based Project Centralization Algorithm . . . 21
2.2 Graph Coloring Based Approach . . . 24
2.2.1 Constraint Graph, Constraint Structure . . . 25
2.2.2 D-graph Representation of a Project Set . . . 27
2.3 Algorithm and Optimal Solution . . . 32
2.4 Experiments on Network Libraries . . . 35
2.5 Case Study in Managing Version Variants . . . 36

3 Process Centralization and Verifying Distributed Applications **40**
3.1 Process Centralization Issues . . . 41
3.2 Implementation . . . 42
3.3 Comparisons of Runtime Performance . . . 44

3.4 Centralization with JPF 46

4 Program Analysis Enhanced Automatic Testing and Testing Multiple Versions 48

4.1 Program analysis enhanced random testing 49

4.1.1 Introduction and Motivation 49

4.1.2 Weakness Found in Randoop 51

4.1.3 Our Enhancements 55

4.1.4 Experiments 60

4.2 Testing Multiple Versions 71

4.2.1 Background and Introduction 71

4.2.2 Recent Related Works 72

4.2.3 Our Approach 74

4.2.4 Case Study 77

4.2.5 Results 78

4.2.6 Discussion 80

4.2.7 Threats to Validity 80

4.2.8 Summary 81

5 Conclusion and Future Direction 82

5.1 Conclusion 82

5.2 Future Directions 84

Chapter 1

Introduction

This chapter gives an overview and introduction of this thesis. We present the basic concepts and background that our research work is based on. Section 1.1 introduces the issues on multiple versions and the motivation of our work. Section 1.2 gives a general introduction to multiple versions and version conflict issues. We present related existing techniques and compare them to our proposed solutions in the later of this thesis. Section 1.3 introduces the challenges in the analysis and verification of distributed systems, involving multiple versions, multiple components, multiple processes, network communications and concurrencies. Section 1.4 introduces automatic software testing and the limitation of existing techniques. It also discusses our enhanced techniques and novel solutions for testing multiple versions, where testing results can be shared among multiple versions to improve overall performance. Section 1.5 presents the outline of this thesis. Finally, section 1.6 summarizes the main contributions of this thesis.

1.1 Background and Motivation

Nowadays, software becomes one of the most important medium in information and communication technology to improve the labor force. It is widely used to for many important tasks in industry to manage data, control vehicle engines, schedule

network communication, manipulate medical equipment. It affects almost every aspect of our society and our daily life. Software defects and failures, however, can cause severe tragedy and loss. Therefore, ensuring software quality becomes very important.

Formal verification and testing are two major approaches to detect software defects and to improve software quality, obeying its specification. They occupy a large part of the cost in software development and maintenance. However, existing verification and testing techniques support only to analyze a single version of a product. As the recent widely adoption of revision control system and Software Product Lines (SPLs), and *clone-and-own* (widely adopted in software industry) techniques, more and more similar version variants of a software project are created. This causes new challenges in the verification and analysis of these version variants which share many common codes with differences for each version. Separate verifications and analyses for each individual version would cause many redundancies in analyzing the common codes. As the result, we cannot share the analysis results among different versions, failing to improve the overall performance.

In practice, the multiple version variants also cause version conflicts, bringing program failures. Such conflicts usually happen in a component based system such as distributed systems and cloud systems, where each component is developed and maintained separately. Changes during the life-cycle of components require the coexistence of multiple versions. Many languages and platforms designed to support only one version also create challenges in representing and executing multiple versions. Managed languages like Java and C# only support loading each version of a class once by a class loader. This further creates challenges in analyzing multiple versions. If each version variant is separately analyzed, it causes redundancies such as storage and runtime memory. The analysis is also incomplete without analyzing possible interactions between different versions. For example, the whole semantic space of a distributed application contains the interaction and communication between different peers. They are inherently not supported by existing techniques. It is a challenge to analyze the software systems that are allow the coexistence of multiple versions.

In this thesis, we propose a general technique to manage multiple versions, that shares common codes and resolves the version conflict while preserving the behavior of each version. Based on this general approach, we further perform studies and pro-

pose novel verification and testing techniques to solve many real-world problems of many softwares that have multiple version issues, such as the version conflict (allowing the coexistence of multiple versions) and multiple version analysis duplications (reduce redundancies in analyzing the common code).

To solve these challenging issues of multiple versions, we first propose and implement the framework *project centralization*. It represents multiple version variants as a single product, resolving the version conflict, sharing their commonality as much as possible while preserving the behavior and version space of each version. We formalize the version conflict problem and proposes a graph based approach to calculate the optimal/near-optimal solution. We implement this approach and demonstrate its effectiveness by evaluating our method on many practical applications. We further demonstrate its effectiveness in software verification and testing software with multiple versions.

The project centralization builds a foundation for our further studies on proposing the techniques and tools for the verification of distributed applications with the coexisting of multiple versions (one of current most challenging issues in software verification) and advanced automatic testing for multiple version variants by reusing testing results.

The distributed applications also involve multiple processes, creating the further issue to preserve the semantics of each process in original distributed applications. We propose process centralization techniques that transform multiple processes into a single centralized process. The combined project centralization and process centralization allow existing verification tools to work on distributed applications, where multiple versions and processes are involved.

Due to state explosion issues of existing verification techniques (model checking), they are usually used to solve medium size application with concurrencies such as distribution applications, where traditional testing techniques are difficult to cover the overlapping semantic spaces caused by concurrency. For large and practical industrial software applications, testing is still one of the most important techniques. Manually producing these test cases is laborious with many defect difficult to be detected. Automatic testing techniques like random testing are proved to be useful in finding bugs and improve software quality. However, exiting automatic techniques suffer from low code coverage, which are unable to uncover many program paths, leaving many defects never be covered.

We further propose our program analysis enhanced automatic testing techniques. Our technique is fully automatic. It extracts the domain knowledge (both at static phase and run-time phase) from the software under test to guide run-time test case generation. Our tools are carefully evaluated on more than 30 practical widely used software applications and inside SCCH, demonstrating that it represents the state of the art in automatic testing (randomized) techniques. Based on this solution, we propose novel testing techniques for multiple versions to share common testing results of each version and improve the overall testing performance. Although many automatic testing solution for a single versions exist, to the best of our knowledge, no previous solution can automatic testing multiple products (more than 3) to share their testing results and improve overall performance (code coverage). We are the first to propose automatic testing and sharing results from multiple versions to improve overall improvements.

In summary, this thesis intends to propose a general technique and framework for managing multiple versions. Based on this, we further propose novel verification and testing techniques, both improving existing work and proposing new methods for the verification and testing software multiple versions that are not handled previously. Our studies and proposed techniques and tools give the first set of solutions to the challenging issues of revealing software defects and improving software quality in the era of multiple versions.

In the rest of this section, we first summarize our major research contributions. Then, we introduce some basic concepts, related work, and our concerns on each issues for multiple versions. While our implementation supports programs written in Java, the concepts and techniques presented in this thesis generalize to other managed languages and platforms.

1.2 Version Conflict Issues and Project Centralization

A software system continuously changes to increase its functionalities, to fix bugs, and to adapt to new requirements over its life cycle [52]. Such changes are released as a sequence of updated revisions. Some related product variants are also created by coping and modifying existing ones (the clone-and-own approach), which is a

common practice for developing new software products [74]. As many similar product variants are developed, effective management and analysis of these products become very important. Separately managing each of these products wastes storage by code duplication and causes redundancies for analysis and verification. For example, if multiple software products have an identical function, independently executing the same test case for such a function for each product causes redundant executions [75], because it shows the same implementation and routine behavior in each occurrence. On the other hand, version conflict occurs when multiple versions have the functions of the same name, but of different implementations and of runtime behaviors. This is typically observed when multiple variations of a software interact interact like component based application and distributed system. The analysis and verification of these applications also cause a challenge in handling these multiple versions for analysis tools and execution platforms like Java and C#, which are designed to handle only single version.

Hnetynka et al. [36] originally discussed the component version conflict problem in Java component-based systems. They adopt the renaming approach by augmenting the class name of each component with a version identifier in dynamic class loading. However, such a trivial renaming solution is not scalable as it can not share common codes. Loading many classes into a VM will decrease its runtime performance. Another approach adopts a modified non-standard Java VM [22, 37, 84] that allows loading multiple versions of the same named classes multiple times. However, it does not share common codes. A non-standard VM does not give correctness guarantee, either.

To share common codes at runtime, Paal et al. [65] have proposed a customizable hierarchical class loader approach to separate the version space so that two common code of version variants that share the same system class loader share all the code loaded by the system class loader. However, this approach requires manually configuring the hierarchy of class loaders. It also lacks flexibility in controlling the class dependency and loading orders. Such a class loader approach does not separate the Java core library space either [55].

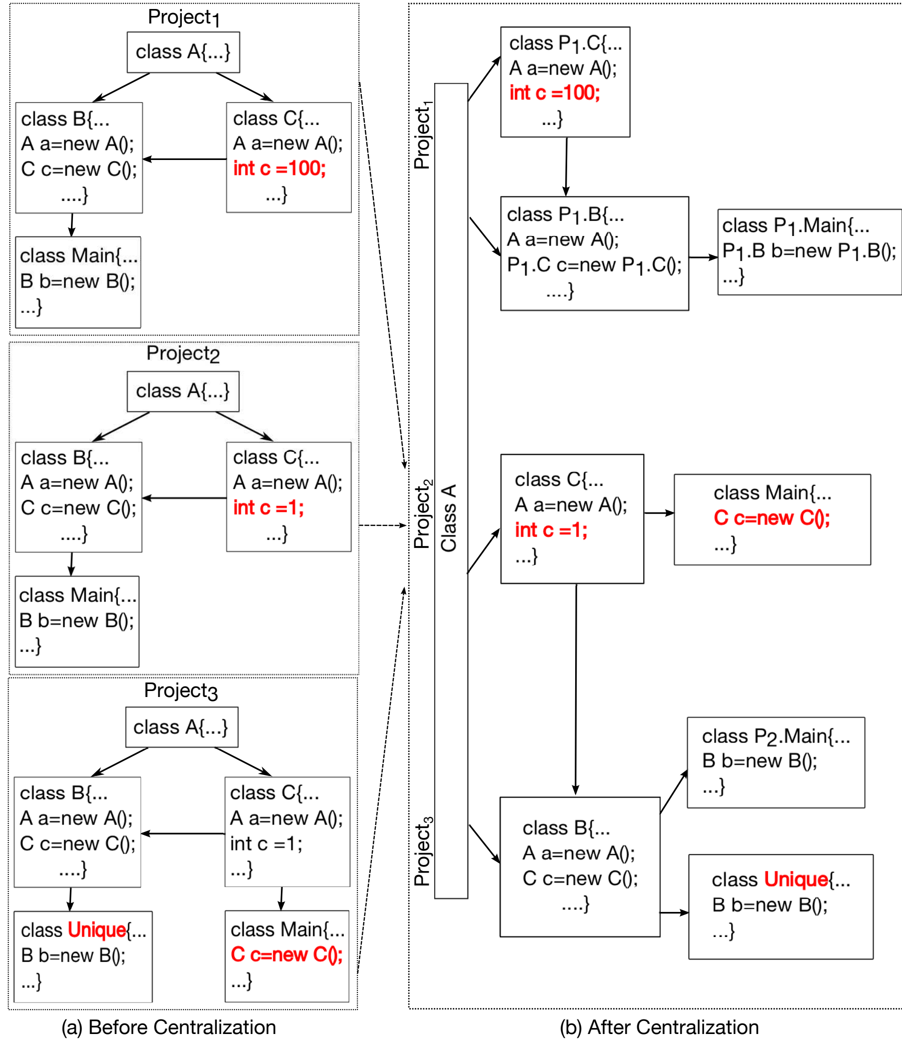
Deduplication is a general approach to address redundancy in memory contents at run-time [96]. Deduplication shares identical memory blocks in virtualized execution environments. This approach is independent of target languages and platforms and has recently been extended to sharing contents of similar (but not identical)

memory blocks as well [33]. However, it is less specific and less efficient than the project centralization based approach, and is not amenable to program analysis as it is agnostic of the structure of the underlying data.

Compared with these strategies, other strategies have also been proposed to manage multiple variants of a software system in the past decades. Some researchers advocate refactoring them into Software Product Lines (SPL) [27,62,80], and more researchers manage them in a revision control system [2,3,63], where software merging techniques are often used. However, none of such approaches preserve behavior of each version variant which must be satisfied for verification and analysis so that the the solution is sound. A difficulty of refactoring multiple similar product variants into an SPL is to extract the commonality and variability that are usually represented as a feature model. However, this needs domain analysis to identify features and establish the connections between a feature and its corresponding code, which proves to be difficult to automate and therefore lacks accuracy. Although the family based products generation and representation of an SPL can ease some static analysis, the dynamic verification and testing still have to be applied to each version variant separately. On the other hand, as a main challenge of software management using a revision control system, we must resolve version conflicts when merging existing products. A version control system usually adopts a text-based comparison to track changes so that different types of documents can be handled. The unawareness of underlying language structure hinders further analysis for the multiple product variants by existing tools. Although some language structure-aware merging strategies have been developed to handle different languages, these techniques do not guarantee to preserve the behavior of each product [2,3,63]. Therefore, such as approach cannot be used for analyzing multiple product variants either.

We propose a project centralization approach by transforming multiple products into a single one. It manages multiple versions to avoid code redundancies while preserving the behavior for each of them. Our technique shares common codes whenever possible while preserving the behavior of each product and resolving their version conflict for analysis. Our goal is to build a general analysis and verification framework that can handle multiple versions.

Figure. 1.1 gives an example of a project centralization, which we use in the rest of this thesis. Each project consists of a set of classes and represent a version variant of a product. We draw a directed edge from class cl_1 to cl_2 if cl_2 depends



(a) Before Centralization (b) After Centralization

Figure 1.1: Project Centralization Example.

on cl_1, which we use to represents the class dependency. For example, we draw a directed edge from class A to C in Project$_1$ because class C references A. In Fig. 1.1(a), Project$_1$ and Project$_2$ can share most of their classes except C, where we can see that different versions are used. Compared to Project$_2$, Project$_3$ has a different version of class $Main$ and a new class $Unique$.

Project centralization transforms multiple projects into a single one, in which each project preserves its version space while sharing common code whenever possible. Fig. 1.1(b) shows the centralization result for projects in Fig. 1.1(a). All projects share class A. Project$_1$ renames its classes to $P_1.C$, $P_1.B$, and $P_1.Main$

to separate the version space. Similarly, Project_2 and Project_3 share classes C and B, and Project_2 renames its class *Main* to $P_2.Main$. Classes *Main* and *Unique* in Project_3 are left unchanged. The centralized result preserves the behavior of each project.

A trivial solution would entail renaming all classes and duplicating all code for each project. However, code duplication consumes more storage to represent the code repository and large runtime memory by loading more classes. This causes this naive approach difficult to scale up to larger applications. In other words, this approach cannot reduce the redundancies in program analysis and verification either. For example, when analyzing a distributed system containing 20 peers, duplicating all projects from these peers is not necessary as they can reuse some shared classes with proper transformation, saving both storage and runtime memory. Therefore, it is worth of thought to share the common class codes.

Our goal is to resolve the class version conflict where necessary while sharing equivalent classes among projects. Figure. 1.1.(b) shows a project centralization result without duplicating the code that can be shared. The trivial solution produces 13 classes. However, it is only necessary to keep one version of class A after project centralization. Similarly, we can keep two versions of class B and C. One version is shared by Project_2 and Project_3, and the other version is used for Project_1. Actually, Figure. 1.1.(b) shows an optimal solution, where only 9 classes are needed. Detailed formalization and illustration of project centralization are discussed in Section 2.

1.3 Distributed System and the Challenge in its Verification

The term *distributed application* contains three aspects [10]: firstly, it means an application whose functionalities are split into a set of cooperating, interacting functional units. Each unit runs as a process that has its internal state (data) and operations to manipulate the state. Secondly, these functional units can be assigned to different machines. A single machine, however, may host several functional units at the same time. Finally, the functional units communicate with each other through network.

On modern operating systems, distributed applications are implemented as a

system using multiple processes. They usually run on different hosts and communicate over a network. Nowadays, the distributed and cloud software systems play a very important role in both industry and our daily life. Most non-trivial software applications are implemented as distributed, networked applications. Therefore, developing techniques and tools to improve the software quality of distributed systems are very important. However, the verification and analysis of the distributed and cloud system are even more challenging. Multiple processes run concurrently and use asynchronous communication over network. Activities of processes can be arbitrarily interleaved and no two executions of the same application need to be identical. Such nondeterminism from concurrency makes the run-time behavior of distributed application difficult to understand, predict, debug, and verify. This problem becomes more exacerbated if multiple threads inside a process are involved because they create concurrency inside a process as well as between processes.

Although many existing tools like Java PathFinder (JPF) [94], Java Interactive Profiler (JIP) [87] work on single-process applications, they do not support multi-process applications. If powerful analysis tools that support a single process were available to multiple processes, development and analysis of distributed systems would become easier.

To ensure the software quality of distributed application and enable existing tools to analyze distributed application, we propose a program transformation technique *process centralization* to separate the process runtime space of each component in distributed application, while preserving its semantics. After such transformations, existing tools like JPF can be directly applied to verify and analyze the whole distributed system automatically.

Stoller [82] initially proposes to the concept centralization for verifying distributed Java applications in Java PathFinder (JPF). Artho et al. [5] improve the accuracy of centralization and implement an automatic tool for verification by JPF. However, their solution cannot support distributed applications with multiple versions. Their implementation uses the outdated SERP bytecode library [78], which makes it unable to work on current Java applications. Their solution also targets JPF and cannot support other analysis tools.

Other work on verifying distributed applications in JPF includes net-iocache [6] and modeling the Java class loader [79]. Both solutions are specific to JPF. Compared with the centralization approach, net-iocache analyzes a single peer of a

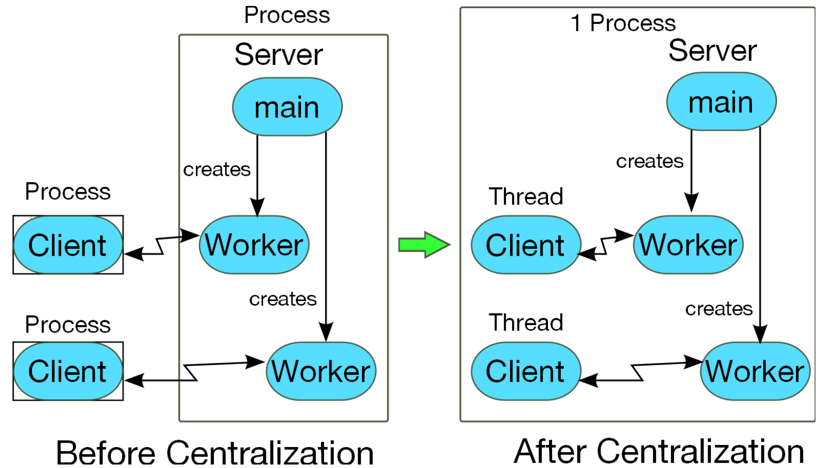


Figure 1.2: Process Centralization Example

distributed application, which runs faster by sacrificing the completeness of verifying all execution traces. Specifically, we can find bugs that net-iocache cannot detect, but that centralization can. As a new feature of JPF v7, modeling multiple processes by using separate class loaders is proposed [79]. It uses class loaders to separate process name spaces by a roundtrip collaboration between JPF and host VM, which is useful to enhance net-iocache. However, the cost of this roundtrip switch between JPF and host VM is expensive, which makes such approaches difficult to scale up for larger applications. Additional works on startup, shutdown behavior preservation, and modeling network library are also necessary to verify distributed applications.

Compared with previous work, we intend to build an automatic centralization tool for general purpose analysis of distributed applications. Figuare. 1.2 shows the process centralization of a distributed application containing three components: one server and two clients. Before centralization, each component runs as a process. Inside the server process, three threads run concurrently. Thread *main* creates two *Worker* threads to separately serve each connected client. After centralization, all processes are wrapped as threads and run as one process. Centralization was initially proposed to exhaustively verify distributed applications. However, a large number of combinational states limit possible analysis to small applications. We propose using centralization for a general (not necessarily exhaustive) analysis of distributed applications.

Centralization enables many distributed applications to be available to existing

tools and reduces the difficulty for analyzing them. For example, in a single-process debugger, a distributed application cannot be paused in a single step; when centralized this becomes possible. Other dynamic verification tools such as Java Race Detector [53] and JCarder [45] detect data races and deadlock bugs for single-process applications. However, they do not support multi-process applications. Meanwhile, profiling tools [40,87] are useful for gathering the runtime performance of distributed applications. They only provide methods to separately analyze each component. This brings additional overhead by creating and destroying multiple VMs and lacks scalability. Because a centralized application runs on one single VM, these profiling tools can collect all the related profiles and scale to larger distributed applications. Finally, visualization tools [47,85] are useful for understanding the runtime behavior of applications. They extract call graphs of distributed applications automatically, which helps to understand how its multiple components interact. Centralization makes it possible to visualize distributed systems also in this case. We will discuss the process centralization and implementation issues in Seciton 3.

As resolving class conflicts is essential for centralizing larger distributed applications, we propose our solution and implement it in our tool. We also improve the process centralization on the transformation issues. Although the large state space of distributed applications limits software model checkers to small cases, our centralization approach enables existing dynamic analysis tools to analyze practical distributed applications.

We will discuss our process centralization solution and implementation issues in Seciton 3.

1.4 Software Testing for Multiple Version Variants

Software testing has long been recognized as one of the most essential and expensive activities in the whole software development cycle. According to [1], around 30% to 90% development efforts are spend on software testing. Especially for complex software system, many software behaviors must be test and the test input selection space is very large.

Unit testing is a widely accepted and important measure in software development. A unit test consists of a sequence of function calls. Object-oriented functions called *methods* operate on an object instance called the *receiver instance.* Each in-

vocation executes a specific path, which depends on the program state. Currently, most industrial testing code (e.g. at Microsoft, Google) is manually written.

However, Manually crafting test sequences is a labor-intensive task. Random testing automatically generates test sequences to execute different paths in a method under test (MUT) [34]. It randomly constructs object instances as the receiver and input arguments of the MUT. However, we found that existing random techniques suffer from low code coverage. Reasons are that randomly generated sequences may not able to set up the receiver in all the required states, or that the required input arguments for invoking the MUT cannot be generated automatically.

Existing automatic testing techniques suffer low code coverage. To the best of our knowledge, no techniques support to test multiple versions simultaneously (more than 2) and study whether sharing testing results from multiple versions can improve performance. Therefore, our goal is to design novel testing strategy to improve existing techniques for practical applications and to further extend enhanced testing techniques for multiple versions.

There exists a large body of work on automated test case generation [11, 18, 19, 21, 28, 43, 59, 66, 100]. In this section, we discuss work that is closely related to our approach.

For more related work on automatic test case generation techniques and tools, we refer readers to representative papers [1, 29, 61, 68], which give a more thorough introduction.

1.4.1 Variants of Random Testing

The critical step in automatic test case generation for object-oriented programs is to prepare the input objects with desirable object states. An input object can be constructed by either direct construction [11,59] or method sequence construction to return the desired types [66, 88, 101]. Direct construction approaches like Korat [11] and TestEra [59] construct objects by directly assigning fields. However, these approaches require specifications defined in languages like JML or Alloy, so they are not fully automated.

Most existing random techniques create the required input objects by *method sequence construction* [21, 66, 67, 88, 100]. JCrasher [21] creates input objects by using a parameter graph (similar to our method dependency graph in Section 4.1.3.4)

to find method input and return type dependencies. Eclat [67] and Randoop [66] use feedback from previous tests to generate future tests. These approaches are closest to ours.

Bounded exhaustive approaches [11, 59, 97] generate method sequences exhaustively up to a given length. However, real-world applications may sometimes require long sequences, making it difficult to determine a limit.

Adaptive random testing (ART) [16] is another variant of random testing. It selects test inputs evenly across the input space. ART has been shown to improve the efficiency of random testing and to reduce the number of tests required to reveal the first error. ARTOO [19] extends the ideas of ART to object-oriented languages by defining object distances based on types and their matching fields. However, ART faces the challenge of high-dimensional input spaces, which limits the usage of ART to larger software.

Like most related tools, our enhancements conform to the declared access modifiers by not calling private or protected methods. Previous work reveals that directly testing these methods (by changing non-public modifiers to public) increases code coverage [42]. However, such tests are difficult to validate. By convention, a public method should throw an exception if a precondition is violated.[1] Non-public methods may also use assertions, which normally indicate an internal flaw in the software, and thus a failed test.[2]

1.4.2 Use of Domain Knowledge

Existing test cases contain information on valid test sequences. To explore more valid sequences and generate more diverse object states, MSeqGen [88] mines frequently used sequence patterns from the code base to guide sequence generation. RecGen [101] performs lightweight analysis on fields accesses by different methods and favors testing those methods that access the same fields together. Palus [100] performs dynamic analysis from provided sample test cases to train method invocation models, and static analysis to identify method relevance based on field accesses. Both results are used to guide run-time test case generation. OCAT [43] adopts object-capture and replay techniques, where object states are captured from the

[1] `http://www.oracle.com/technetwork/java/effective-exceptions-092345.html`

[2] `http://docs.oracle.com/javase/7/docs/technotes/guides/language/assert.html`

running sample test cases, and then used as input for further testing. Yet other work modifies Randoop to generate oracles for regression tests [72].

Evolutionary approaches [7,28,30,90,90] like Evosuite take a test suite and evolve new test sequences from it, trying to generate new sequences with diverse object states.

Compared to these techniques, our approach does not depend on existing test cases written by a human, which are often biased towards certain types of test sequences [14].

1.4.3 Symbolic Execution

Symbolic execution represents input as symbolic values and executes the program based on abstract semantics, computing path conditions based on input parameters by leveraging constraint solvers. Tools like Java PathFinder [95] and Symbolic PathFinder [56] generate test cases this way. Hybrid approaches of random (concrete) and symbolic execution, called *concolic* execution, are implemented by tools like DART [31], Cute and JCute [76, 77], and Pex [89].

Our approach uses a light-weight static analysis instead of symbolic techniques to approximate path conditions.

1.4.4 Automatic Testing Techniques for Multiple Versions

With the more widely application of revision control system and Software Product Lines, more similar product variants would be created. Therefore, our goal is to develop a testing strategy and tool that can work for multiple product variants, sharing their testing results and improving the testing performance like cod coverage.

However, to the best of our knowledge, none of existing testing approaches is able to test multiple software projects simultaneously. The key issue is to share the testing results among multiple projects and reduce the testing redundancies. A previous work [43] has demonstrated the testing result from existing test cases of one project (through object serialization) can further increase the testing performance (code coverage) of the same project. This work mentioned the potential challenges in testing results sharing during version update. However, it does not prove the usefulness of a testing result in improving the testing performance of related projects.

Another challenge issue is the redundancies in testing multiple projects, where the redundancies include code redundancies, test case execution redundancies, analysis redundancies, and so on. Furthermore, deciding the testing priority for the code of multiple projects under the given resources (time and cost) constraint is difficult problem.

Although test case augmentation [75, 99] provides a partial solution for generating new test cases for version update. However, it is still challenging to automatically work for multiple versions and accurately identify which test cases can be shared, especially for Software Product Lines with too many version variants, limiting such an approach only to small cases.

Compared to existing works, our goal is to develop an automatic testing framework for multiple versions, sharing testing results to improve overall performance while reducing the testing redundancies among multiple versions. More details are presented in Section 4.

1.5 Thesis Outline

The rest of this thesis is organized as follows. Section 2 first discusses and formalizes the version conflict problem. Then the project centralization based approach are proposed and formalized. A simple project centralization is first given. More advanced graph representation is then proposed the corresponding is also given. Based on this, we discuss the optimal solution and an heuristic algorithm. We evaluate these proposed algorithms on practical software projects to compare their effectiveness. In addition, we explore the usefulness of our proposed approach to manage version variants from software evolution and SPLs. Section 3 discusses the process centralization issues and challenges. We presents our implementation and applied it to verify practical distributed systems with multiple versions. The effectiveness of different project centralization solution based process centralization results are also compared in their runtime performance. Section 4 first presents our program analysis enhanced automatic testing techniques and its evaluations. Then it further presents the improvement of project centralization and its combined tool chain with automatic testing techniques for multiple versions. The evaluations on version variants from both software evolution and Software Product Lines demonstrate its effectiveness, showing that our techniques can reduce testing redundancies, and also

share testing results to improve overall performance. Finally, Section 5 concludes the thesis and points out future challenges and research directions.

1.6 Research Contributions

The contributions of this thesis are summarized as follows:

- **Multiple version management and version conflict resolution:**

  1. We formalize the version conflict problem and classify the classes of each version variant into three categories.
  2. We propose and formalize our project centralization approach to manage multiple product version variants and resolve their version conflict.
  3. We formalize the solutions that can be achieved in representing multiple products while preserving their behaviors. We first propose and implement a worklist based project centralization algorithm. We further propose and formalize a *D-graph* representation for all products under analysis. Based on the formalization, we transform project centralization into the *graph coloring* problem. A corresponding D-graph based project centralization algorithm (optimal solution and heuristic one) is also proposed and implemented. Various experiments are conducted to demonstrate the effectiveness of our approach.
  4. We conduct case studies to explore the effectiveness of project centralization in managing real world software products during software evolutions and Software Product Lines (SPL). Experiments on real world projects are also performed to compare effectiveness of different approaches.

- **The analysis and verification of distributed applications with multiple versions**

  1. We propose and implement *process centralization* transformation approach to separate the runtime space of each process in a distributed application. We summarize the essential issues to preserve the process semantics by program transformation.

2. We implement an automatic process centralization tool. Furthermore, we integrate project centralization and process centralization as a tool chain.
3. We evaluate our tools on real world distributed applications. We apply Java PathFinder (one of the current representative model checking tools for software verification) to our centralized applications and successfully find bugs which cannot be detected by conventional techniques.
4. We thoroughly evaluate and compare the run-time performance of differently proposed centralization techniques and the original application without centralization.

- **Enhanced automatic testing and testing multiple versions**

1. We propose and implement a combined static and dynamic analysis guided random test case generation tools.
2. We demonstrate effectiveness and practical values of improving coverage and bug finding capability on 30 widely used real-world software and software inside Software Competence Center Hagenberg (SCCH).
3. We report the bugs found and get confirmed from several developer groups including Apache, Google to demonstrate the usefulness of our tool in revealing both known and unknown bugs in their products.
4. We further refine the accuracy of project centralization to method level to share more common methods, which is required for automatic test case generation for multiple versions.
5. We implement multiple version testing based on project centralization tool chain, which reduces redundancies in testing common code while sharing the testing results among multiple version to improve overall performance.
6. We evaluate the effectiveness of our techniques on many Software Product Lines and software evolutions, showing that our technique improves the overall performance in testing multiple versions and reducing testing redundancies.

Chapter 2

Project Centralization

The concurrent usage of different versions of a software product is common, especially in a component-based systems, where each component is developed and managed independently. Analyzing such multiple software version variants causes version conflict on a single VM, where each class loader is allowed to load one version of a class. The project centralization resolves possible version conflicts by separating the version space of each component, while sharing common code among different systems.

In this section, first formalize version conflict problem and project centralization. Then, we discuss a simple worklist based algorithm. Furthermore, the D-graph representation is proposed and formalized. The corresponding graph based algorithm is also proposed and discussed. Experiments on real-world version variants compare the effectiveness of these approaches. Finally, we make case study on 3 large benchmarks from software evolution and SPLs to demonstrate the usefulness of project centralization in managing multiple version variants.

2.1 Concepts in Project Centralization

To formally define the version conflict problem, we summarize several important concepts in project centralization and define the optimal solution in project cen-

tralization.

2.1.0.1 Definition of Project Centralization and its optimal solution

A Java class is uniquely identified by its name (including package name) and implementation. For a class cl, we use $cl.name$ and $cl.code$ to denote its class name and implementation, respectively.

Given two classes cl_1 and cl_2, cl_1 and cl_2 are equivalent, denoted by $cl_1 = cl_2$, if they form a **Type-1** clone pair [73], where $cl_1.name$ is identical to $cl_2.name$, and $cl_1.code$ and $cl_2.code$ are also identical except for variations in whitespace, layout and comments.

Definition 2.1.1. A *project* consists of a unique identity and a set of classes, in which each class has a distinct name. Given a project p, we write $\#p$ as the number of classes in p, and denote a class cl in p by $p.cl$. Two projects p and q are *identical*, denoted by $p \equiv q$, if they hold the same set of classes. We write $p \not\equiv q$ if p and q are not identical.

A project represents an abstract view of the class repository of a version component. Each version is represented by a project. Furthermore, the combination of all versions can be represented by one *centralized* project by merging small projects. Two versions may use code from either the same project or different projects. In both cases, code repositories of multiple components can be represented as a centralized project, sharing common code.

We use project to abstract the class repository of a component-based application, that contains multiple components.

Definition 2.1.2. Let p be a project. We define $\text{NAME}(p) = \{cl.name | cl \in p\}$ as the set that contains all class names in p. For a class name $cln \in \text{NAME}(p)$, we define $\text{GetClass}(p, cln) = p.cl$, where $p.cl.name = cln$, as a function to get the class named cln in p. Let P be a set of projects. We define $\text{NAMES}(P) = \cup_{p \in P}\text{NAME}(p)$ as the set containing all the class names in P, and $P \uparrow cln = \{p \in P | cln \in \text{NAME}(p)\}$ as the set of all projects that contain the class named cln.

Definition 2.1.3. *Process centralization* is the transformation of multiple processes into a single one with equivalent runtime behavior.

Previous work [5] assumes all the processes run under the same project, where each class has only one version. To centralize processes containing classes with multiple versions, we propose to perform *project centralization*. Before defining project centralization, we first define project renaming substitution and project equivalence.

Definition 2.1.4. Let p be a project, and cln_1 and cln_2 be two class names. Project *renaming substitution* $p[cln_1/cln_2]$ is defined as a project in which p substitutes its class name cln_1 for cln_2. Substitution includes class names and references to them. A renaming substitution $p[cln_1/cln_2]$ is a *normal substitution* if $cln_1 \notin \text{NAME}(p)$ and $cln_2 \in \text{NAME}(p)$.

Definition 2.1.5. Let p_1 and p_2 be two projects; p_1 is equivalent to p_2, denoted by $p_1 = p_2$, if they can be renamed to identical projects by normal substitutions It is not difficult to prove that this is an *equivalence* relation that is reflexive, symmetric, and transitive.

Definition 2.1.6. *Project centralization* transforms a set of projects $P = \{p_1, p_2, \ldots, p_n\}$ P into one single project p_{centr} such that $\forall p \in P. \exists p' \subseteq p_{centr}. p = p'$. We denote all the centralized results of P that satisfy this condition by $\text{CENTR}(P)$.

Project centralization requires preservation of the class version space for each project. Each component-based application that runs as the original project can also run as the centralized project with the same runtime behavior. The projects to be centralized can either be different versions of a component or different components.

We define the class dependency in a project as follows.

Definition 2.1.7. Let cl_1 and cl_2 be two classes in a project p. Class cl_1 depends on cl_2, denoted by $cl_2 \rightarrow cl_1$ if $cl_1.code$ references $cl_2.name$.

Given two classes cl_1,cl_2, $cl_1 \rightarrow cl_2$ represent that cl_1 depends on cl_2 so that if cl_1 is renamed, all references of cl_1 in cl_2 must also be renamed to preserve the behavior.

Let P be a set of projects to be centralized. To separate the version space of each project, we classify the classes of a project $p \in P$ into the following categories:

1. *Unique Class.* $\text{UNIQUE}(p, P) = \{cl \in p | \forall q \in P \backslash p.\ cl.name \notin \text{NAME}(q)\}$. A unique class of project $p \in P$ has a unique name across all projects in P.

2. *Conflict Class.* CONFLICT(p, P) =$\{cl \in p | \exists q \in P.\ cl.name \in (\text{NAME}(p) \cap \text{NAME}(q)) \wedge p.cl \neq \text{GetClass}(q,\ cl.name)\}$. The name of a conflict class appears in multiple projects, including p, but with different implementations.

3. *Shared Class.* SHARED(p, P) = $\{cl \in p | \exists q \in P \backslash p.\ cl.name \in (\text{NAME}(p) \cap \text{NAME}(q)) \wedge p.cl = \text{GetClass}(q,\ cl.name)\}$. A shared class of p shares both its name and implementation with other projects in P.

In our example in Fig. 1.1(a), classes A and B are shared classes in all projects. The cases for classes C and *Main* are more complex: class C is a conflict class in Project$_1$, but it is both shared and a conflict class in Project$_2$ and Project$_3$. Similarly, class *Main* is a conflict class in Project$_3$, and it is both shared and a conflict class in Project$_1$ and Project$_2$. Informally, a version conflict happens when we try to make project centralization on a project set, whose element contains conflict classes.

Definition 2.1.8. Let P be a set of projects to be centralized. Centralized project p_{centr} is minimal (optimal) if $p_{centr} \in \text{CENTR}(P)$ and $\forall p'_{centr} \in \text{CENTR}(P)$. $\#p_{centr} \leq \#p'_{centr}$.

Consider a general scenario of centralizing a set of projects $P = \{p_1, p_2, \ldots, p_n\}$. There may exist multiple solutions that satisfy *Definition* 2.1.6. Among these solutions, the optimal solution outputs the minimal number of classes. which is also the optimal solution to solve version conflict and share the common code among all projects in P. If $\forall p_i \in P. \text{CONFLICT}(p_i, P) = \emptyset$, the optimal result is the union of all projects in P, $\bigcup_{i=1}^{n} p_i$. If conflicts are present, $\exists p \in P. \text{CONFLICT}(p, P) \neq \emptyset$, the goal is to separate all conflict classes in P while maximizing the sharing of classes.

2.1.1 A Worklist Based Project Centralization Algorithm

The main issue of resolving version conflict is to properly separate the class version for each project. This entails renaming the conflict classes and all their references to separate their versions. However, such renaming may cause shared classes not shareable anymore, as their internal references to other classes are renamed differently across projects. Consider the example in Fig. 1.1: Project$_1$ and Project$_2$ can share class B before project centralization. They have to rename their class C to

Algorithm 1 Worklist Based Project Centralization Algorithm

```
1: procedure SimpleProjectCentralization
     Input: A project set P = {p_1, p_2, ..., p_n}
     Output: A renamed project set P' = {p'_1, p'_2, ..., p'_n},
              where ∀i ∈ {1, ..., n}.p_i = p'_i ∧ Conflict(p'_i, P') = ∅
2:     for i ← 1, n − 1 do
3:         P ← P/p_i
4:         worklist w ← ∅
5:         queue q ← ∅
6:         w ← Conflict(p_i, P)
                                   ▷ add the conflict classes of p_i into worklist
7:         q ← w                   ▷ add each element of w to q for renaming
8:         while w ≠ ∅ do
9:             Pick and Remove cl from w
10:            for all cl' ∈ depends(cl, p_i) do
11:                if cl' ∈ Shared(p_i, P)
12:                    ∧ cl' ∉ q then
13:                    q.enque(cl')
14:                    w ← w ∪ {cl'}
15:                end if
16:            end for
17:        end while
18:        p'_i = renameProject(q, p_i)
19:                  ▷ make normal renaming substitution of p_i for all classes in q
20:    end for
21:    P' ← ∪_{i=1}^{n} p'_i
22: end procedure
```

a different name to solve version conflict, though. After that step, B cannot be shared anymore as it references C. Therefore, it is necessary to rename the conflict classes and propagate their renaming effect in each project.

Alg. 1 gives our initial solution that uses a worklist based algorithm to propagate the renaming effect. It renames all the conflict classes and propagate and rename all those shared classes until no conflict classes exist.

The input of this algorithm is a set of projects to be centralized. The output is the renamed projects containing no conflict classes, and each of them is equivalent to the project before renaming. Given a project set P with $\#P = n$, the algorithm iterates and renames each of the first $(n-1)$ projects. We use the worklist w for traversing the class dependency relation, and the queue q for storing the classes needing renaming, respectively.

For each project, the algorithm first calculates all the conflict classes of the current project and put them into q for renaming. For each conflict class, its renaming effect then propagates to all the shared classes. The renaming effect fully propagates until the worklist w becomes empty. After finding all the classes needing renaming, renameProject(q, p_i) in Fig. 1 performs normal renaming substitution on project p_i according to the renaming queue q.

Each class of a project p_i is added to the worklist at most once and only those classes that are either shared or conflicting can be added to the worklist. The output condition is also guaranteed to hold. There is no class version conflict because all conflict classes and their propagation effect are resolved. In addition, projects before and after renaming are equivalent by normal substitution.

For complexity, we consider analyzing a project set P with $\#P = n$ which includes m class names in total. All input projects are internal data structures that represent class raw files. The class classifications and dependency relations are pre-calculated during the preprocessing phase. The complexity for checking the existence of a class in set $\text{Conflict}(p, P)$ or $\text{Shared}(p, P)$ is $\mathcal{O}(m)$. The dependency relation set $\text{DEPENEDS}(cl, p_i)$ for class cl in project p_i contains at most $(m-1)$ classes (excluding the class self dependency). In the worst case, the complexity for traversing the class dependency relation in the loop of worklist is $\mathcal{O}(m^2)$. Therefore, the complexity for calculating the renaming decision of project set P is $\mathcal{O}(m^2 \cdot n)$. After class renaming, no two projects hold conflict classes and all projects can be centralized into one project by taking the union of all their classes.

This algorithm correctly separates the version space of all input projects. However, it does not always output a satisfactory solution. The limitation of this algorithm is caused by a lack of version linkage of classes among all projects. It renames a class as long it is a conflict class even though a conflict class is still possible to be shared.

The optimized solution separates the project version space while sharing classes whenever possible. A class in a project can be both a conflict class and a shared class. This algorithm does not distinguish a conflict class and a class that is both shared and conflicted. It simply renames the class as long as it is a conflict class. Consider the example in Fig. 2.1(a). P is a project set to be centralized, where $P = \{p_1, p_2, p_4, p_3\}$ and each project $p_i \in P$ has one class A. There exist two versions of A in P, where p_1, p_2 hold one version, and p_3, p_4 hold the other version. We represent different versions of a class by different colors. The simple algorithm renames all A's in p_1, p_2, p_3, but not in p_4, resulting three classes after project centralization as shown in Fig. 2.1(b). However, an optimized solution produces only two classes (the two versions of A): one is shared by p_1 and p_2 and the other is shared by p_3 and p_4 as shown in Fig.2.1(c). The limitation of this algorithm is caused by a lack of version linkage of classes among all projects. We present our improved

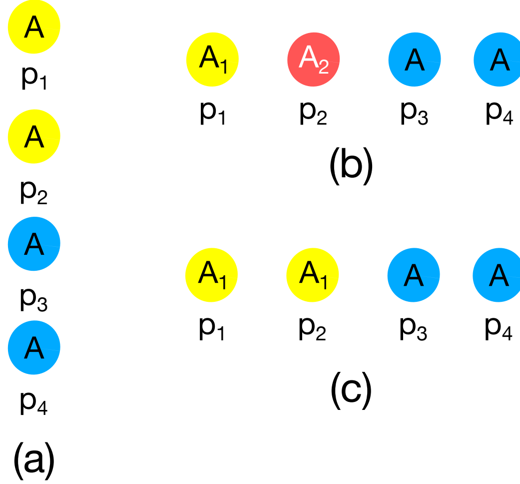


Figure 2.1: Worklist based Algorithm Example

solution in next section. Suppose there exists a class A in $P' = \{p'_1, p'_2, \ldots, p'_n\}$ with n versions. Theoretically, renaming any $(n-1)$ versions of A can resolve version conflict. However, properly selecting the $(n-1)$ versions for renaming is a difficult problem. Whenever a class is renamed, its effects propagate in the project, which may result in more classes that must be renamed. To obtain an optimal solution, we need to search all possible version combinations of classes in all projects for renaming. When centralizing a project with m names each with n versions, the complexity for searching the optimal combination of class renaming actions is $\mathcal{O}(n^m)$. Algorithm in Fig. 1 approximates this by simply renaming all classes that are both shared and conflicting.

2.2 Graph Coloring Based Approach

To obtain the optimal solution of project centralization to share more common code, we further propose a *D-graph* representation for projects and transform project centralization into a graph coloring problem. Project centralization is an optimization problem. The goal is to obtain a centralized project with the minimal number of classes under given version constraints. We formalize the *D-graph* representation for projects and transform project centralization into a graph coloring problem. Then, we present a corresponding algorithm based on existing graph coloring solutions. In the rest this section, we arbitrarily fix a set of projects P for centralization.

2.2.1 Constraint Graph, Constraint Structure

A *constraint structure* represents a graph node of a D-graph. Each constraint structure contains a name to represent all classes with the same name. It also contains a *constraint graph* represents the version relation of all classes with the same name in P. In a constraint graph, the nodes are all projects that contains the classes with that node name. We use the edge to represent the version relations of all classes with the same name from different projects so that two classes are connected if they are different versions. Given a project set P, we can initially built all D-graph nodes by analyze all classes and their relations.

We show that all constraint graphs of a class name in P form a complete lattice, and extend the constraint graph to a *constraint structure*, which represents a node of a *D-graph*, as defined below.

Definition 2.2.1. Let *cln* be a class name in P. A constraint graph of *cln* in P consists of a pair of node set $P\!\uparrow\!cln$ and edge set CE, denoted by $\langle P\!\uparrow\!cln\ , CE\rangle$, such that if there exist two projects $p, p' \in P\!\uparrow\!cln$ and $(p, p') \in CE$, p and p' cannot share the class named *cln*.

Each node in a constraint graph of name *cln* is a project containing a class named *cln*. Two project nodes that are connected by an edge, cannot share the same version of the class named *cln*. Edges in a constraint graph are undirected; given any two project nodes m and n, (m, n) and (n, m) represent the same edge. Let $G = \langle P\!\uparrow\!cln\ , CE\rangle$ be a constraint graph of *cln* in P, and P' be a project set. We write the subgraph of G to P' as $\langle P'', CE'\rangle$ (denoted by $G\!\uparrow\!P'$), where $P'' = P' \cap P$, and CE' is a restriction of CE to P''.

Given a class name *cln*, it is not difficult to observe that there exist multiple constraint graphs of *cln* that satisfy *Definition* 2.2.1, sharing the same node set but differing in their edge sets. We define their least upper bound and partial order relation, respectively.

We define the partial order relations over constraint graphs and write $\mathbb{CG}_{cln}$ as the constraint domain of *cln* in P, respectively.

Definition 2.2.2. We define $\sqsubseteq_{cg} \in \mathbb{CG} \times \mathbb{CG}$ as a binary relation such that for any two constraint graphs $G_1, G_2 \in \mathbb{CG}$ with $G_1 = \langle P_1, CE_1\rangle$ and $G_2 = \langle P_2, CE_2\rangle$, $G_1 \sqsubseteq_{cg} G_2$ if $P_1 \subseteq P_2$ and $CE_1 \subseteq CE_2$. We denote the least upper bound of G_1 and G_2 by $G_1 \sqcup_{cg} G_2 = \langle P_1 \cup P_2, CE_1 \cup CE_2\rangle$.

It is not difficult to prove that $\sqsubseteq_{cg} \in \mathbb{CG} \times \mathbb{CG}$ is a partial order and $(\mathbb{CG}, \sqsubseteq_{cg})$ is a partially ordered set.

Definition 2.2.3. Let $G = \langle P\!\uparrow\!cln\ , CE\rangle$ and $G' = \langle P\!\uparrow\!cln\ , \ CE'\rangle$ be two constraint graphs of a class name *cln*. We define their least upper bound as $G \sqcup_{cg} G' = \langle P\!\uparrow\!cln\ , CE \cup CE'\rangle$, which is also a constraint graph of *cln*. We denote the constraint graph domain of *cln* by $\mathbb{CG}_{cln}$, which contains all the constraint graphs of *cln* in P.

Definition 2.2.4. Let $\mathbb{CG}_{cln}$ be the constraint graph domain of the class name *cln* in P. We define $\sqsubseteq_{cg} \in \mathbb{CG}_{cln} \times \mathbb{CG}_{cln}$ as a binary relation such that $\forall G_c, G'_c \in \mathbb{CG}_{cln}.\, G = \langle P\!\uparrow\!cln\ , \ CE\rangle \wedge G' = \langle P\!\uparrow\!cln\ , CE'\rangle \wedge CE \subseteq CE' \Rightarrow G \sqsubseteq_{cg} G'$.

It is not difficult to prove that $\sqsubseteq_{cg} \in \mathbb{CG}_{cln} \times \mathbb{CG}_{cln}$ is a partial order and $(\mathbb{CG}_{cln}, \sqsubseteq_{cg})$ is a partially ordered set.

Let *cln* be a class name in P; we write $\mathbb{CG}_{cln}$ as the subdomain of $\mathbb{CG}$, where $\mathbb{CG}_{cln} \subseteq \mathbb{CG}$ and $\forall G \in \mathbb{CG}_{cln}.\, G = \langle P', CE'\rangle \Rightarrow P' = P\!\uparrow\!cln$. It is not difficult to prove that the partially ordered set $(\mathbb{CG}_{cln}, \sqsubseteq_{cg}, \sqcup_{cg}, \sqcap_{cg},\ \bot_{cg}^{cln}, \top_{cg}^{cln})$ is a complete lattice [8] with, $\forall X \subseteq \mathbb{G}$, $X = \{x_1, x_2, \ldots, x_n\}$:

- a least upper bound $\sqcup_{cg} X = x_1 \sqcup_{cg} x_2 \sqcup_{cg} \ldots \sqcup_{cg} x_n$,
- a greatest lower bound
  $\sqcap_{cg} X = \sqcup_{cg}\{y | \forall x \in X.\, y \sqsubseteq_{cg} x\}$,
- a least element $\bot_{cg}^{cln} = \langle P\!\uparrow\!cln\ , \emptyset\rangle$,
- a greatest element $\top_{cg}^{cln} = \langle P\!\uparrow\!cln\ , CE_{greatest}\rangle$, where $CE_{greatest} = \{(m, n) | n, m \in P\!\uparrow\!cln \wedge m \neq n\}$.

We extend the constraint graph to a *constraint structure.*

Definition 2.2.5. A *constraint structure* CS in a project set P consists of a name in NAMES(P) (denoted by $CS.name$) and a constraint graph of the name $CS.name$ (denoted by $CS.CG$). We write $\langle CS.name, CS.CG\rangle$ for the structure.

We define the partial order relation and least upper bound for constraint structures.

Definition 2.2.6. Let CS_1 and CS_2 be two constraint structures. We define the partial order relation (constructed from the partial order of the constraint graph) $CS_1 \sqsubseteq_{cs} CS_2$ if $CS_1.name = CS_2.name$ and $CS_1.CG \sqsubseteq_{cg} CS_2.CG$. The least upper bound of CS_1 and CS_2 is defined as $CS_1 \sqcup_{cs} CS_2 = \langle CS_1.name, CS_1.CG \sqcup_{cg} CS_2.CG\rangle$ if $CS_1.name = CS_2.name$.

For a class name *cln* in P, it can be shown that all constraint structures that share *cln* also form a complete lattice.

Definition 2.2.7. Given a constraint graph set $\mathbb{G}_c$ of a project set P and a class name *classname*, $\forall G_c, G'_c \in \mathbb{G}_c$ where $G_c =< P, E_c >$ and $G'_c =< P, E'_c >$, we define least upper bound of G_c and G'_c as $G_c \sqcup G'_c =< P, E_c \cup_c E'_c >$. We generalize the definition of least upper bound to a set. $\forall X \subseteq_c \mathbb{G}_c$ where $X = \{x_1, x_2, \ldots, x_n\}$, the least upper bound of set X is defined as $\sqcup_c X = x_1 \sqcup_c x_2 \sqcup_c \ldots \sqcup_c x_n$.

For a constraint graph set $\mathbb{G}_c$ of a project set P and a class name *classname*, it is not difficult to prove that the partially ordered set $(\mathbb{G}_c, \sqsubseteq_c, \sqcup_c, \sqcap_c, \bot_c, \top_c)$ forms a complete lattice [8] such that $\forall X \subseteq \mathbb{G}_c$ with:

- a least upper bound $\sqcup_c X$.
- a greatest lower bound $\sqcap_c X = \sqcup_c\{y|\forall x \in X, y \sqsubseteq_c x\}$.
- a least element $\bot_c =< P, \emptyset >$.
- a greatest element $\top_c = G_c$, where G_c is the complete graph over node set P.

2.2.2 D-graph Representation of a Project Set

We formalize the *D-graph* representation for projects, and propose constraint equations to calculate the minimal D-graph that satisfies version constraints. We then transform project centralization into a graph coloring problem. After explaining all the *D-graphs* of a project set forms a complete lattice, we show the calculation procedure to get minimal D-graph that satisfies all version constraints of classes in the project set. Based on this, we propose a graph coloring based project centralization algorithm and discuss its optimal solution.

We show that all D-graphs of a given project set form a complete lattice, constructed by the tensor product of constraint structures. Based on this representation, we prove the NP-completeness to calculate the optimal solution of the project

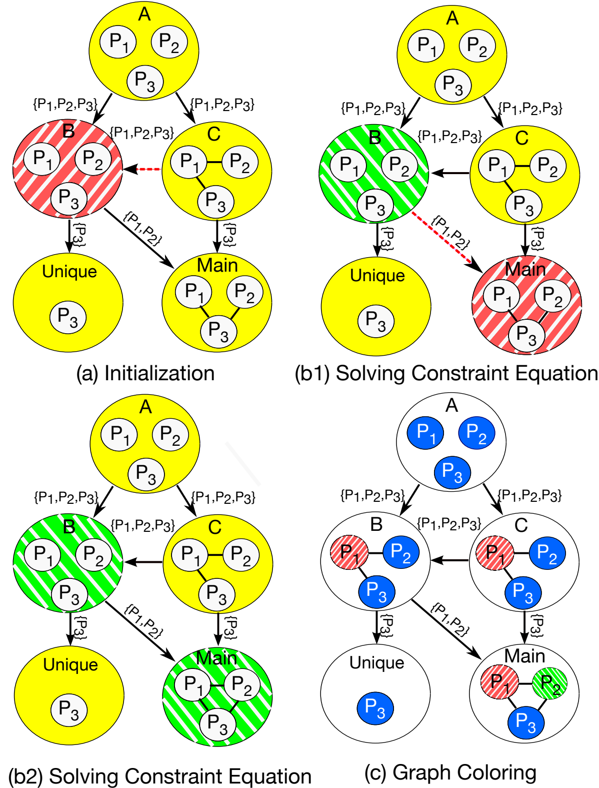


Figure 2.2: D-graph Representation Example

centralization for version separation by transforming it into the graph coloring problem.

Definition 2.2.8. A *D-graph* of a set of projects P consists a node set N of constraint structures and an edge set E (denoted by $\langle N, E\rangle$) with each edge $e = (l, m) \in E$ associated with a set of projects $e.set = \{p \in (P\uparrow l.name \cap P\uparrow m.name)$ $|\text{GetClass}(p, l.name) \rightarrow \text{GetClass}(p, m.name)\}$ such that:

1. Name set $\{n.name|n \in N\}$ is the same as NAMES(P).
2. $\forall i, j \in N. e = (i, j) \wedge e.set \neq \emptyset \Rightarrow e \in E$.

Let $G = \langle N, E\rangle$ be a D-graph of P. Each node $n \in N$ is a constraint structure that represents all versions of the classes with that node name $n.name$. Its constraint graph is used to keeps the version relation of these classes. For two nodes

$m, n \in N$, the existence of an edge (m, n) from m to n entails that the classes named $m.name$ and $n.name$ have a dependency relation in a project $p \in P$, and p have both classes named $m.name$ and $n.name$. Edges in E are directed: (n, m) and (m, n) are different edges.

Fig. 2.2(a) gives the corresponding initial D-graph of the project set in Fig. 1.1(a). The larger node is the constraint structure node, inside which its name and constraint graph are shown. For example, the node named A with its constraint graph indicates that its name exists in projects P_1, P_2 and P_3. No edge exists between these projects, meaning all these projects initially have the same version of class named A. The label of an edge in a D-graph shows the projects in which the two constraint structure nodes connected by that edge have a dependency relation. For example, the edge from node B to *Main* indicates that classes named B and *Main* have a dependency relation in both P_1 and P_2.

We next define the *underlying graph* of a *D-graph*.

Definition 2.2.9. Let $G = \langle N, E \rangle$ be a D-graph of P. We define its *underlying graph* as the graph of G by ignoring the constraint graph of each constraint structure node in N, denoted by $|G| = \langle |N|, E \rangle$, where $|N|$ represents the nodes of G that ignore all their constraint graphs.

The underlying graph $|G|$ for a project set P is unique. There are multiple D-graphs that share $|G|$, differing in their constraint structures. We use $\mathbb{G}_{|G|}$ to represent the domain all D-graphs of P such that they share $|G|$ as the underlying graph. We simply write $\mathbb{G}$ if $|G|$ is clear from context. We continue to define a partial order over $\mathbb{G}$ and show that all its D-graphs also form a complete lattice.

Given a node $n \in N$, we use $\mathbb{N}$ to represent the node set that share the same $|n|$.

Definition 2.2.10. Let $\mathbb{G}$ be a D-graph domain of P, and $G = \langle N, E \rangle$ and $G' = \langle N', E \rangle$ be arbitrary two D-graphs in $\mathbb{G}$. G and G' have the binary relation $G \sqsubseteq G'$ if and $\forall n \in N. \forall n' \in N'. n.name = n'.name \Rightarrow n \sqsubseteq_{cs} n'$. The least upper bound of G and G' is $G \sqcup G' = \langle N'', E \rangle$, where $N'' = \{m \sqcup_{cs} n | n \in N \wedge m \in N' \wedge m.name = n.name\}$.

Assuming NAMES$(P) = \{cln_1, cln_2, \ldots, cln_k\}$, the constraint graph domain and underlying graph of P be $\mathbb{G}$ and $|G| = \langle |N|, E \rangle$, respectively. It is not difficult

to prove that $(\mathbb{G}, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ (constructed by the Cartesian product all constraint structures in N) is a complete lattice [8].

such that $\forall X \subseteq \mathbb{G}$ where $X = \{x_1, x_2, \ldots, x_n\}$ with:

- a least upper bound $\sqcup X = x_1 \sqcup x_2 \sqcup \ldots \sqcup x_n$,
- a greatest lower bound $\sqcap X = \sqcup\{y | \forall x \in X. \, y \sqsubseteq x\}$,
- a least element $\bot =$ $\langle\{\langle cln_1, \bot_{cg}^{cln_1}\rangle, \langle cln_2, \bot_{cg}^{cln_2}\rangle, \ldots, \langle cln_k, \bot_{cg}^{cln_k}\rangle\}, E\rangle$,
- a greatest element $\top =$ $\langle\{\langle cln_1, \top_{cg}^{cln_1}\rangle, \langle cln_2, \bot_{cg}^{cln_2}\rangle, \ldots, \langle cln_k, \top_{cg}^{cln_k}\rangle\}, E\rangle$.

For a directed graph G, we refer $\mathbb{G}$ as the set of directed graphs that share $|G|$. Since the possible constraint graphs of each node in G form a complete lattice, $< \mathbb{G}, \sqsubseteq \sqcup, \sqcap, \bot, \top >$ also forms a complete lattice, which is constructed by the tensor product of the constraint graphs of all nodes [8] such that $\forall X \subseteq \mathbb{G}$ where $X = \{x_1, x_2, \ldots, x_n\}$ with:

- a least upper bound $\sqcup X = x_1 \sqcup x_2 \sqcup \ldots \sqcup x_n$.
- a greatest lower bound $\sqcap X = \sqcup\{y | \forall x \in X, y \sqsubseteq x\}$.
- a least element $\bot =< \bot_c^{x_1} \times \bot_c^{x_2} \times \ldots \times \bot_c^{x_n} >$.
- a greatest element $\top =< \top_c^{x_1} \times \top_c^{x_2} \times \ldots \times \top_c^{x_n} >$

Correct project centralization requires separating the version spaces of each project by renaming. Renaming a class also entails renaming all references to it accordingly. We use *conflict edges* to represent version constraint conditions. Such constraints capture the effect that different versions of a class are not separated due to the version separation of another class that this class depends on.

Definition 2.2.11. Let $G = \langle N, E\rangle$ be a D-graph of P. Let $e \in E$ be an edge and $e = (m, n)$, where $m.CG = \langle P \uparrow m.name \ , \ CE\rangle$ and $n.CG = \langle P \uparrow n.name \ , \ CE'\rangle$. The edge e is a *conflict edge* if $\exists p, q \in e.set. \, (p, q) \in CE \wedge (p, q) \notin CE'$.

An edge $e = (m, n)$ in a D-graph of P is a conflict edge, if there exist two projects p, p' such that they are connected by an edge in $m.CG$ but not in $n.CG$.

For example, the dashed edge from node C to B in Fig. 2.2(a) is a conflict edge. P_1 and P_3 must hold a different version of class C, as they are connected by an edge in $C.CG$. This entails renaming their C to a different name; furthermore, as C is referenced in the code of B, P_1 and P_3 must also be connected in $B.CG$ to separate version space. To ensure the correctness of project centralization, all such version constraints must be resolved.

Definition 2.2.12. A D-graph $G = \langle N, E \rangle$ of P is *valid* if there does not exist an edge $e \in E$ such that e is a conflict edge.

Correct project centralization requires finding a valid D-graph given a set of projects, such that all version constraints are resolved. An optimal solution requires a D-graph that is both valid and minimal. This entails propagating the minimal version constraints so that each constraint structure node n in that D-graph satisfies the equations in (2.2.1), where $\text{IN}(n)$ and OUT(n) are the incoming and outgoing constraint conditions (represented by the constraint graphs) of node n, and $\text{IN}_0(n)$ and $\text{OUT}_0(n)$ are the corresponding initial conditions, respectively. The functions in equational system (1) are monotonically increasing over complete lattices with finite height. Therefore, a minimal solution exists and can be computed by iterating the equation system [86] until reaching its least fixed point.

$$
\boxed{
\begin{aligned}
&\text{IN}(n) = \Big(\bigsqcup_{\substack{cg \\ m \in \text{Pred}(n)}} \text{OUT}(m) \Big) \uparrow (m, n).set \\
&\text{OUT}(n) = \text{IN}(n) \sqcup \text{OUT}(n) \\
&\text{IN}_0(n) = \emptyset \\
&\text{OUT}_0(n) = n.CG
\end{aligned}
}
\tag{2.2.1}
$$

The constraint equations defines the minimal constraint for each node to separate the project version space correctly. Initially, the incoming constraint for each node $n \in N$ is empty and the outgoing constraint equals $n.CG$. The constraint graph $n.CG$ is initialized during the construction of the D-graph such that two project nodes are connected by an edge in $n.CG$ if they hold a different version of the class named $n.name$. The initial D-graph of the example in Fig. 1.1(a) is depicted in Fig. 2.2(a). The constraint equations are solved iteratively until no conflict edge exists. Fig. 2.2(b1) and Fig. 2.2(b2) show such steps to solve constraint equations for node B and *Main*, respectively. The minimal valid result is given in Fig. 2.2(c).

The remaining task is to ensure no two nodes connected by an edge in a constraint graph of the minimal valid D-graph share the same version. This is equivalent to coloring the graph such that two nodes connected by an edge are colored differently. After coloring, all the nodes in a constraint graph with the same color can share the same version of a class, and nodes colored differently cannot share the same class and should be renamed accordingly. In our example, all three projects share class A; class B outputs two versions, one of which is for P_1 and the other version is shared by P_2 and P_3 as shown in Fig. 2.2(c). Therefore, we transform the project centralization problem into a graph coloring problem.

The optimal version separation is to find the chromatic number k for each constraint graph in the directed graph. All the nodes in a constraint graph with the same color can share the same version of a class. Given a constraint graph G_c, we denote $MinimalColorOutput(G_c)$ as the minimal number of colors to ensure no two connected points share the same color.

2.3 Algorithm and Optimal Solution

For a project set P, the optimal solution of project centralization is to separate the version space of each project while keeping share the common classes as many as possible. It entails to output the least number of classes while keeping the version space of each project separate.

Given a constraint graph G, we denote MinimalColorOutput(G) as the minimal number of colors to ensure no two nodes connected by an edge share the same color.

Definition 2.3.1. Given a directed graph $G =< N, E >$ of project set P, G is an optimal graph iff there exists no conflict edges in G, and for any directed graph G' of P without conflict edges, where $G \neq G' \wedge |G| = |G'|$, $\sum_{n \in N} \text{MinimalColorOutput}(n.CG) < \sum_{n' \in N'} \text{MinimalColorOutput}(n'.CG)$

Based on the D-graph representation for a set of projects, obtaining the project centralization solution for version separation entails the following steps:

1. Solve the constraint equation for each node to get the minimal valid D-graph.

2. Color the constraint graph in each constraint structure node n of the D-graph such that any two nodes connected by an edge in $n.CG$ are colored differently.

Algorithm 2 Graph Coloring Based Project Centralization

```
1: procedure ProjectCentralization
   Input: A set of projects P = {p_1, p_2, ..., p_n}
   Output: The centralized project p_centr,
           where ∀p ∈ P. ∃p' ⊆ p_centr. p = p'
2:   DGraph ← ∅
3:   nameSet ← collectName(P)                          ▷ Collect all class names
4:   DGraph.nodeSet ← ∅
5:   for all name ∈ nameSet do                         ▷ Build a node for each name
6:     DGraph.nodeSet ← DGraph.nodeSet
                        ∪ {createNode(name, P)}
7:   end for
8:   for all src ∈ DGraph.nodeSet do                   ▷ Add edges
9:     for all targ ∈ DGraph.nodeSet\src do
10:      tempSet ← {p|p ∈ (P↑src.name ∩ P↑targ.name)
11:        ∧GetClass(p, src.name) → GetClass(p, targ.name)}
12:      if tempSet ≠ ∅ then
13:        (src, targ).set = tempSet
14:        Dgraph.edgeSet ← Dgraph.edgeSet ∪ {(src, targ)}
15:      end if
16:    end for
17:  end for
18:  Initialize IN(n) and OUT(n) for each n ∈ DGraph.nodeSet
19:  SCCs ← calculateSCC(DGraph)
20:  TopoSCCs ← calculateTopologicalOrder(SCCs)
21:  Dgraph ← EquationSolver(Dgraph, TopoSCCs)   ▷ Alg. 3 solves constraint equations until reaching the least
       fixed point
22:  for all node ∈ DGraph.nodeSet do
23:    graphColoring(node.CG)
                      ▷ Color each output constraint graph by existing algorithm
24:  end for
25:  p_centr ← NormalRenaming(DGraph)
                      ▷ Perform normal substitution according to the coloring results
26:  return p_centr
27: end procedure
```

3. Perform normal renaming substitution for each constraint graph such that project nodes with the same color still share the same class after renaming while nodes with different colors do not.

We propose a project centralization algorithm based on graph coloring (see Alg. 2). Given a set of projects as input, the algorithm first initializes the nodes and edges of the D-graph (lines 2–17), and the IN and OUT constraints for each of its nodes. To improve convergence towards the fix point, we calculate all Strongly Connected Components (SCC) and sort them in a topological order. Next, function EquationSolver (see Alg. 3) is called to solve the constraint equations iteratively for the ordered nodes until reaching the least fixed point (lines 18–21). The last step colors the constraint graph of each node and performs normal renaming substitution (lines 22–25).

Alg. 2 is guaranteed to terminate. For the complexity of our algorithm, we assume the D-graph is initialized and it is only necessary to calculate the renaming

Algorithm 3 Solve Constraint Equations

```
1:  function EquationSolver
                    ▷ Solve constraints for a given graph in SCCs' topological order
    Input: graph: a D-graph,
           TopoSCCs: the topological order of SCCs for graph
    Output: graph: the minimal valid D-graph
2:    for SCC ∈ TopoSCCs do
                    ▷ Visit each SCC in topological order
3:      repeat      ▷ Repeat if constraints of a node in SCC change
4:        for n ∈ graph.nodeSet ∧ n ∈ SCC do
5:          IN(n) ← (⨆_{m∈Pred(n)} OUT(m)) ↑(m, n).set
6:          OUT(n) ← IN(n) ⊔ OUT(n)
7:          n.CG ← OUT(n)
8:        end for
9:      until ∀n ∈ (graph.nodeSet ∩ SCC). IN(n) and
                  OUT(n) do not change
10:   end for
                    ▷ Function terminates when no constraint conditions change
11:   return graph
12: end function
```

decision for further processing. The D-graph initialization (lines 2–17) and renaming substitution (line 25) are specific to the given projects and operating system, so we do not consider them in the complexity analysis.

Let P be the input projects with $\#P = n$, and its initial D-graph be $G = \langle N, E \rangle$ with $\#N = m$ and $\#E = l$. It iteratively solves constraint equations for each node of G in order until reaching a fix point. As the constraint functions in Alg. 3 are monotonically increasing over a complete lattice with finite height, the least fixed point can be reached in no more than $n^2 \cdot m$ iterations. The complexity of computing the constraint conditions for a node is $\mathcal{O}(n^2 \cdot m)$. Computing the SCCs and their topological cost $\mathcal{O}(m \cdot l)$. Assuming the complexity of adopted graph coloring algorithm for a k-vertex and t-edge graph is $\alpha(k, t)$, the total complexity of solving a given D-graph and making the renaming decision is $\alpha(n, l) \cdot m + \mathcal{O}(n^4 \cdot m^3)$. The complexity of the iterative framework analysis depends on the graph structure of a given D-graph. Proving a tighter upper bound for complexity is beyond the scope of this paper and is future work.

To obtain the optimal project centralization result, it is necessary to apply an exact graph coloring algorithm on the achieved minimal D-graph so that the number of classes in the output is minimal. We adopt an existing optimal algorithm [9], which solves the problem of a k-vertex graph in PSPACE and in time $\mathcal{O}(5.283^k)$. However, the exact graph coloring is an NP-complete problem and its complexity is exponential. Therefore, we also provide the option to use the Greedy Independent Sets (GIS) coloring approach [50] with complexity $\mathcal{O}(k \cdot v)$, where k and v represent

the number of vertexes and edges, respectively.

2.4 Experiments on Network Libraries

We have implemented the proposed algorithms in Java and applied them on seven real-world Java projects as benchmarks. Our implementation transforms the Java bytecode of the target applications. As our tool does not require the source code of the application, it also works for languages other than Java that compile to Java bytecode. Table 2.1 summarizes the benchmarks which are all network libraries that can be used as a component of a distributed application. The project size and number of classes are listed. All experiments in this thesis were run on an Intel Core i7 Mac 2.4 GHz with 8 GB of RAM, running Mac OS X 10.8.3 and Oracle's Java VM, version 1.7.0_21.

To quantify and compare the effectiveness of each algorithm in sharing common code, we define *Shared* as the ratio of shared classes to output classes: $Shared = \frac{\#ClassShared}{\#OutputClass}$, we also use S.F. for simplicity. A class named *cln* is counted as shared if at least two projects share that class in the renaming decision. Consider the graph coloring results in Fig. 2.2(c), classes A, B, C are shared because multiple projects are colored the same in their constraint graph, but class *Unique* and *Main* are not. *Shared* ranges from 0 to 1; the larger its value, the more classes are shared. The trivial renaming approach renames all classes of each projects and shares no classes, *Shared* is therefore 0. We run the experiment to centralize projects of each benchmark with a different number of instances per version. Each experiment is repeated 60 times to collect the *Shared* value, run time, and storage saving ratio (project size before centralization/after centralization). After choosing a benchmark with a specific setting, the *Shared* value and storage ratio of a given project centralization algorithm are unique. As for the run time, we discard the data of the first 10 runs, which may be influenced by disk I/O to read the input, and take the average of the other results.

The overall experimental result is summarized in Table 2.2. We have five experimental settings for each benchmarks with two versions of a project, from centralizing one and two project instances, to seven instances of both versions. The performance of the three approaches for each setting is listed in columns *Shared*, Storage Ratio, and Time, respectively, which can be compared horizontally and vertically. To

interpret the data, we take the project centralization of Edtftpj-2.3.0 and Edtftpj-2.4.0 as an example. we apply three approaches on each We have five settings for each benchmark, from centralizing one and two project instances, to seven instances of both Edtftpj-2.3.0 and Edtftpj-2.4.0. For example, on the setting in row five, the optimal algorithm and greedy algorithm only use 17.0% of the total storage but the simple algorithm uses 43.1%. We can draw the same conclusion for other benchmarks in Table 2.2.

For each experimental setting, the greedy coloring approach outperforms the simple solution and performs as well as the optimal solution in sharing common code and saving storage, as shown by *Shared* value and the storage ratio. As the number of project instances for centralization increases in each benchmark, the $Shared$ value of the simple algorithm decreases. It indicates that some classes are not sharable by the simple solution when centralizing more class instances, but they still can be shared by the greedy solution and the optimal solution. The storage saving ratio of each approach increases as there is a growth in the number of project instances. Compared with the simple solution, the greedy solution and optimal solution both have a larger value of storage saving ratio, meaning that they saves more storage than the simple solution. The greedy approach and the optimal approach are both more effective in sharing common code than the simple algorithm. As for run time, the simple algorithm is the most efficient one, and the optimal algorithm does not scale. Consistent with the complexity analysis in Section 2.2, the run time of the optimal algorithm grows exponentially in the number of projects in Table 2.2. The optimal algorithm cannot solve larger settings in a reasonable time (1 hour). Compared with the other two approaches, the greedy centralization is effective in sharing common code and efficient in practice.

Table 2.1: Summarization of Bechmarks

Project name / version	Edtftp		Ganymed-ss2		Jsmpp		Kryonet		Mime4j-core		Xnio		Netx	
	2.3.0	2.4.0	build209	build210	2.0	2.1	2.08	2.20	0.7.1	0.7.2	2.0.0CR2	2.1.0CR1	0.4	0.5
Bytecode size[KB]	352	391	305	345	457	458	206	252	154	154	249	254	240	246
#Cl. (*.class)	106	113	115	133	201	202	79	104	61	61	72	74	91	88

2.5 Case Study in Managing Version Variants

In this section, we present the case studies we have conducted to investigate the feasibility of managing product variants from software are evolutionary and Software

Table 2.2: Experimental Results of Project Centralization

Project name / version		Inst.		Shared [%]			Strorage Ratio			Time [ms]		
				Simple	Greedy	Optimal	Simple	Greedy	Optimal	Simple	Greedy	Optimal
Edtftpj-2.3.0	2.4.0	1	2	69.3	69.3	69.3	1.79	1.79	1.79	1.07	7.65	10.65
Edtftpj-2.3.0	2.4.0	2	2	53.1	69.9	69.9	1.70	2.35	2.35	1.51	9.71	51.00
Edtftpj-2.3.0	2.4.0	3	3	43.0	69.9	69.9	2.00	3.52	3.52	2.80	11.72	139.06
Edtftpj-2.3.0	2.4.0	5	5	31.1	69.9	69.9	2.32	5.87	5.87	6.77	20.70	5349.42
Edtftpj-2.3.0	2.4.0	7	7	24.4	69.9	N.A.	2.50	8.22	N.A.	12.60	32.30	> 1 h
Ganymed-ss2-build209	build210	1	2	76.0	76.0	76.0	1.94	1.94	1.94	1.02	8.85	12.02
Ganymed-ss2-build209	build210	2	2	61.3	76.0	76.0	1.91	2.54	2.54	1.38	9.65	52.97
Ganymed-ss2-build209	build210	3	3	51.4	76.0	76.0	2.30	3.81	3.81	2.60	12.49	138.04
Ganymed-ss2-build209	build210	5	5	38.8	76.0	76.0	2.75	6.35	6.35	6.39	20.06	5073.64
Ganymed-ss2-build209	build210	7	7	31.1	76.0	N.A.	3.01	8.88	N.A.	12.16	32.13	> 1 h
Jsmpp-2.0	2.1	1	2	89.4	89.4	89.4	2.53	2.53	2.53	1.18	17.68	23.13
Jsmpp-2.0	2.1	2	2	80.8	89.4	89.4	2.92	3.37	3.37	1.88	20.79	83.28
Jsmpp-2.0	2.1	3	3	73.7	89.4	89.4	3.86	5.06	5.06	3.99	28.39	188.76
Jsmpp-2.0	2.1	5	5	62.7	89.4	89.4	5.20	8.43	8.43	10.79	44.76	7109.86
Jsmpp-2.0	2.1	7	7	54.6	89.4	N.A.	6.11	11.80	N.A.	21.67	67.16	> 1 h
Kryonet-2.08	2.20	1	2	58.1	58.1	58.1	1.56	1.56	1.56	0.95	7.45	17.48
Kryonet-2.08	2.20	2	2	41.5	62.6	62.6	1.40	2.02	2.02	1.48	9.83	48.52
Kryonet-2.08	2.20	3	3	32.1	62.6	62.6	1.61	3.02	3.02	2.42	13.40	144.54
Kryonet-2.08	2.20	5	5	22.1	62.6	62.6	1.83	5.04	5.04	4.55	26.39	5119.54
Kryonet-2.08	2.20	7	7	16.9	62.6	N.A.	1.94	7.06	N.A.	8.59	42.94	> 1 h
Mime4j-core-0.7.1	0.7.2	1	2	98.4	98.4	98.4	2.88	2.88	2.88	0.84	3.89	5.74
Mime4j-core-0.7.1	0.7.2	2	2	96.8	98.4	98.4	3.70	3.84	3.84	0.92	4.41	24.94
Mime4j-core-0.7.1	0.7.2	3	3	95.3	98.4	98.4	5.35	5.77	5.77	1.58	5.32	45.27
Mime4j-core-0.7.1	0.7.2	5	5	92.4	98.4	98.4	8.31	9.61	9.61	3.56	6.72	1797.52
Mime4j-core-0.7.1	0.7.2	7	7	89.7	98.4	N.A.	10.90	13.45	N.A.	6.65	10.6	> 1 h
Netx-0.4	0.5	1	2	57.5	57.5	57.5	1.60	1.60	1.60	0.98	8.63	15.46
Netx-0.4	0.5	2	2	45.7	65.4	65.4	1.52	2.13	2.13	1.34	8.68	45.49
Netx-0.4	0.5	3	3	36.0	65.4	65.4	1.77	3.19	3.19	2.18	10.20	130.22
Netx-0.4	0.5	5	5	25.2	65.4	65.4	2.05	5.32	5.32	4.49	18.02	4790.91
Netx-0.4	0.5	7	7	19.4	65.4	N.A.	2.19	7.45	N.A.	8.28	29.10	> 1 h
Xnio-2.0.0CR2	2.1.0CR1	1	2	51.4	51.4	51.4	1.51	1.51	1.51	1.43	4.28	12.33
Xnio-2.0.0CR2	2.1.0CR1	2	2	35.2	54.9	54.9	1.35	2.01	2.01	1.54	5.37	42.03
Xnio-2.0.0CR2	2.1.0CR1	3	3	26.6	54.9	54.9	1.52	3.01	3.01	2.24	9.84	131.10
Xnio-2.0.0CR2	2.1.0CR1	5	5	17.9	54.9	54.9	1.70	5.02	5.02	4.76	17.65	4804.70
Xnio-2.0.0CR2	2.1.0CR1	7	7	13.4	54.9	N.A.	1.78	7.02	N.A.	8.31	28.58	> 1 h

Product Lines by project centralization. The motivation of this case study is to show the effectiveness of project centralization and its potential usage for analyzing multiple version variants without redundancies. The case study shows the usefulness of our approaches as the basic for further study.

The overview of product variants in our case studies is summarized in Table I, which contains both version variants and product variants from a SPL. Column two gives brief descriptions of product variants for each software system by enabling and disabling some features. Column three shows the lines of source code (without comments and whitespaces) for each product. Column four and five list the product size and number of classes (*.class files), respectively.

DrJava [24] in our first case study is a lightweight development environment for writing Java programs. We use its most recent ten version variants released in the last four years. In the second and third case studies, we use the product variants generated from existing software product line ArgoUML-SPL [4, 20] and Health-Watcher [15, 35], respectively. ArgoUML-SPL is the software product line for the UML modeling tool ArgoUML, and HealthWatcher is a cloud computing ap-

plication that manages health records and complaints. Experimental results, where project centralization algorithms are applied to all the selected products of a software project, are summarized in Table II. For each input project set, we give its total number of classes (*.class files), storage usage, and average dependency in columns three, four and five, respectively. We compare the simple algorithm and greedy algorithm in the number output classes, S.F., storage saving ratio, and algorithm execution time and total time. We show these results in columns OUTPUT #Class, S.F., Storage, Alg.Time and Total Time, respectively.

According to the input metrics, DrJava and ArgoUML are both large-size projects. Although the number of input classes for DrJava is twice as large as in ArgoUML, the average dependency for classes in DrJava is much larger. In all experimental settings, the greedy algorithm performs better by outputting fewer classes, sharing more common code and saving more storage. Compared with the simple algorithm on each project set, the greedy algorithm shares 21.9%, 22.9% and 35.2% more classes as indicated by column S.F. The optimized algorithm also saves 22.0%, 12.9% and 29.2% more storage indicated by column Storage. However, the algorithm execution time and total time of the simple algorithm is faster. As the greedy algorithm needs to resolve conflict edges iteratively in the D-graph, the execution time depends on the average dependency between input project sets. Its execution time becomes slow when the number of input classes and average dependency in the input project set are large. From these case studies, we concludes that our greedy algorithm is efficient enough to handle large project sets, and effective in sharing common code and saving storage.

Pro.	DrJava Product Description	LOC	SIZE(KB)	#cl.
P1	Release 20130901-r5756	98793	11895	3936
P2	Release 20120818-r5686	89786	11895	3925
P3	Release 20110822-r5448	89736	11886	3925
P4	Release 20100913-r5387	88401	11659	3877
P5	Release 20100816-r5366	88275	11655	3877
P6	Release 20100711-r5314	87556	11572	3837
P7	Release 20100507-r5246	87258	11499	3824
P8	Release 20100415-r5220	85622	11506	3792
P9	Release 20090821-r5004	81239	9798	3251
P10	Release 20090803-r4975	81044	9711	3242
Pro.	ArgoUML Product Description	LOC	SIZE(KB)	#cl.
P1	All optional feature enable	120348	5147	1915
P2	All optional feature disable	82924	3669	1494
P3	Only Logging disabled	118189	5018	1915
P4	Only Cognitive disabled	104029	4431	1678
P5	Only Sequence Diagram disabled	114969	5033	1881
P6	Only Use Case Diagram disabled	117636	5032	1874
P7	Only Deployment Diagram disabled	117201	5024	1882
P8	Only Collaboration Diagram disabled	118769	5086	1896
P9	Only State Diagram disabled	116431	5008	1880
P10	Only Activity Diagram disabled	118066	5036	1897
Pro.	HealthWatcher Product Description	LOC	SIZE(KB)	#cl.
P1	Base - no extensions applied	5288	261	90
P2	Command pattern applied	5646	273	94
P3	State pattern applied	6112	302	106
P4	Observer pattern applied	6222	309	108
P5	Adapter pattern applied	6379	314	110
P6	Abstract Factory pattern applied	6417	318	114
P7	Adapter pattern applied	6441	319	118
P8	Abstract Factory pattern applied	6468	321	122
P9	Evolution- new functionality added	7709	389	134
P10	Exception Handling applied	7591	389	137

Table 2.3: Version Variants from Software Evolution and SPLs

Project	Algorithm	INPUT			OUTPUT	Alg. Time	Total Time	S.F.	Storage
		#cl.	SIZE(KB)	A.D.	#cl.	(ms)	(ms)	(%)	(%)
DrJava	Simple	37486	113076	17199	29194	6920	10949	6.7	82.2
	Greedy				22247	37166	40063	28.6	60.2
ArgoUML	Simple	18312	48484	3121	6399	2273	3286	27.7	43.0
	Greedy				4485	6959	7706	50.6	30.1
HealthWatcher	Simple	1133	3195	373	771	1128	1242	10.9	82.1
	Greedy				448	1513	1585	46.1	53.8

Table 2.4: Project Centralization Results and Comparison

Chapter 3

Process Centralization and Verifying Distributed Applications

In this section, we summarize our process centralization solution and its application to enable existing tools to analyze and verify distributed application with multiple versions. Experiments with JPF demonstrate that out approach enables existing tools to verify a distributed application with multiple versions, showing that some defects can be found with centralization that are missed with single-process analysis. We first give a general discussion of process centralization issues. Then the implementation process centralization tool is discussed. Experiments on real world distributed applications , where applying centralization tool to network benchmarks, prove the effectiveness of tool automation, showing that the centralized application is more efficient in terms of run time and memory compared with the counterpart without centralization. Finally, we perform two groups of experiments to show the runtime performance of centralization program and verify real world distributed applications by model checkers.

3.1 Process Centralization Issues

This section summarizes the problems that have to be solved to implement centralization of distributed applications correctly. The term *process centralization* is defined as follows.

Definition 3.1.1. *Process centralization* is the transformation of multiple processes into a single one with the equivalent runtime behavior.

We refer the *centralized program* as the program after centralization. Centralization must preserve the semantics of original program. For each execution in the original program, there exists an execution trace in centralized program with the same behavior, and vice versa. To satisfy this requirement, the following issues must be solved.

(1) *Version separation.* Each peer of distributed application can consist of multiple components, where each component is developed and managed independently. In software maintenance and evolution, each component of a component-based system needs to be continually changed over its lifetime to improve its functional capability to satisfy the users' requirements [52]. This can result in conflicts between different versions of the same product that are active at the same time. The problem becomes more exacerbated in a distributed system, where installations are duplicated over many peers. Each application is asynchronously updated in a "rolling update". This creates multiple versions of the components in a system, including both their used libraries and application code. Dumitraş et al. [25, 26] point out that most update failures are not caused by a software defect, but by version conflicts during the update procedure where the main code or library code changes. We adopt our project centralization to resolve version conflict for multiple component in distributed applications. The relation of accuracy of project and the runtime performance of the centralized program is studied in Section 3.3.

(2) *Process Memory space separation.* In a multi-process system, the operating system separates the memory spaces of all processes. This separation is absent in the centralized program but can be emulated by program transformation. In Java-like systems, memory space separation is only necessary on static data, which exists once per VM. Static fields and class descriptors are shared as a single instance of a given class. Accessing these data by different processes without proper separation in the centralized program would cause data races. Therefore, centralization should

keep the memory space of each process separate. We discuss our refinement in Section 3.2.

(3) *Runtime behavior*: Startup and shutdown. Centralization wraps each process of an original program as a group of threads and starts them in the same way before the centralization. We denote each group of such threads by a *centralized process* which has the same runtime behavior as its corresponding process before centralization. Multiple centralized processes run on a single VM after centralization. To manage different centralized processes, Stoller [82] proposes a solution by defining the additional class *CentralizedProcess*. Each application in the original program is wrapped as a *CentralizedProcess* by centralization. Each centralized process holds a unique field as the *process ID*, which is used to identify each application at runtime. The main issues are to start centralized processes in a required order and to preserve the shutdown semantics after centralization. For the analysis of network applications, ensuring that a server is initialized before clients try to connect is important. Otherwise, the client exits prematurely after failing to connect to the server. Shutdown semantics [5] concern the termination of the centralized application. In the Java standard library, invoking methods like Runtime.exit and Runtime.halt [41] terminates the entire VM: the first one runs any previously registered *shutdown hooks*, and tasks that free resources during application shutdown; the second one halts the VM abruptly, without freeing any resources [32]. Centralization should preserve the shutdown behavior of the original program by proper transformation.

3.2 Implementation

We implement our process centralization solution as a four-pass transformation tool, which performs bytecode transformation by using the ASM bytecode library [51]. Before centralization starts, the centralizer parses a user-defined script into a Java startup class file which defines how each process starts. The centralizer transforms all the classes of all projects as described in the script, as defined in previous work [5, 82]. After transformation, the centralized program can be executed from the synthesized startup program.

• *Class Statistics.* The first pass reads in the classes from all projects and builds their internal data structures accordingly. Some statistical information like the

number of class files, the size of each project, and the number of static fields, is calculated in this pass. This provides the user with information about the number of modifications during transformation.

• *Project Centralization* The second pass integrates the project centralization implementation from last section. It reads the data structure built in the first pass and performs project centralization. It provides options to use either the simple algorithm in Alg. 1 or the optimized algorithm (or the optimal solution) in Alg. 2. After project centralization, all projects are represented by one single centralized project data structure for further transformation.

• *Static Fields and Class Descriptors.* The third pass transforms static fields and class descriptors as [5, 82]. For static fields, we transform them into arrays and add one extra dimension if the field is an array. We refine the initialization semantics of static fields by analyzing and transforming the static initializer. We also do not transform *static final* fields if they store the immutable data. The *static final* fields that store mutable data are transformed because they can still cause data races when multiple threads access such data. We also transform static fields that are generated by the Java compiler (synthetic fields). Previous work transforms all static fields without such distinctions.

For class descriptors used as locks, they cannot simply be duplicated like static fields [41]. We adopt the proxy lock approach [5]. Whenever a class descriptor is used as a lock, we use the proxy lock instead. The usage of a class descriptor as a lock or for reflection is distinguished by analyzing instructions of the bytecode. If the class descriptor is on the top of current stack frame and the next instruction is *monitorenter*, it uses the class descriptor as a lock to enter the critical region. Otherwise, the class descriptor is used for reflection which should not be transformed.

• *Startup and Shutdown Semantics.* The last pass implements the startup and shutdown semantics. For the startup semantics, the main issue is to ensure the centralized processes start up in the desired order such that dependencies between them are satisfied; for example, a server needs to be ready to accept connection before its clients are started. A previous implementation [5] is specialized for Java PathFinder by modeling the Java network library. It does not work for other tools than Java PathFinder. We perform instrumentation to a few key network functions. Whenever a component application tries to connect to a port, it creates an external process to check the port status. If the port is open, it continues to connect,

otherwise it waits until the port is open. This approach does not modify the Java network library and can be applied to tools other than Java PathFinder. Shutdown semantics require that a process that calls Java library methods Runtime.exit and Runtime.halt to terminate, only terminates all its threads while other processes may continue running. This requires killing all its threads belonging to the centralized process. In Java, a simple way for a thread to terminate itself is to throw an exception of type *ThreadDeathException*. For the shutdown hooks and allocated resources, we perform code instrumentation so that when a process calls to exit, it call its shutdown hooks and release all its allocated resources.

Table 3.1: Runtime performance comparison

App.	Byted. Size [KB]	#cl. v.1	#cl. v.2	Simple					Greedy					Without Centralization	
				Tran. Time [s]	Trans. Mem. [MB]	Exce. Time [s]	Exce. Mem. [MB]	Stora. [KB]	Tran. time [s]	Trans. Mem. [MB]	Exce. Time [s]	Exce. Mem. [MB]	Stora. [KB]	Exce. Time [s]	Exce. Mem. [MB]
Echo		1	1	0.37	33.59	0.14	22.79	5.35	0.40	36.93	0.14	22.77	5.35	0.27	64.08
Server	2.17	2	2	0.37	34.15	0.14	23.33	6.82	0.41	37.37	0.14	23.02	5.35	0.41	107.82
cl.v1	1.49	4	4	0.38	34.34	0.14	24.15	9.79	0.42	37.93	0.14	24.02	5.35	0.68	195.14
cl.v2	1.49	8	8	0.41	36.27	0.14	25.33	15.70	0.43	38.96	0.14	25.27	5.35	1.22	369.93
Daytime		1	1	0.37	33.24	0.15	24.28	3.90	0.40	36.45	0.19	24.36	3.90	0.36	66.03
Server	1.63	2	2	0.38	33.94	0.16	25.13	5.05	0.41	37.32	0.19	24.88	3.90	0.53	109.60
#cl.v.1	1.16	4	4	0.38	34.27	0.16	26.19	7.34	0.41	37.41	0.19	26.19	3.90	0.87	196.93
#cl.v.2	1.16	8	8	0.40	35.66	0.16	28.62	11.92	0.43	38.56	0.19	28.30	3.90	1.53	371.78
Chat		1	1	0.37	33.91	0.14	23.18	8.39	0.41	37.66	0.14	23.13	8.39	0.63	64.23
Server	3.99	2	2	0.38	34.23	0.15	24.18	10.56	0.42	37.96	0.14	23.91	8.39	0.66	108.30
#cl.v.1	1.98	4	4	0.40	36.09	0.15	25.62	14.93	0.44	38.98	0.14	25.38	8.39	0.72	196.85
#cl.v.2	1.98	8	8	0.44	44.62	0.17	30.12	23.64	0.46	46.50	0.17	28.49	8.39	0.96	373.75
Alphabet		1	1	0.38	35.82	0.42	23.41	8.51	0.42	39.14	0.45	23.53	8.51	0.57	64.60
Server	3.20	2	2	0.40	36.14	0.62	23.83	9.99	0.43	39.19	0.65	23.77	8.51	0.94	108.81
#cl.v.1	3.46	4	4	0.41	38.23	1.02	24.67	12.97	0.45	41.23	1.06	24.54	8.51	1.69	196.95
#cl.v.2	3.46	8	8	0.45	52.45	1.83	26.37	18.91	0.49	57.29	1.87	26.04	8.51	3.14	373.60

3.3 Comparisons of Runtime Performance

In this section, we perform experiment to apply our tool on actual networked applications to compare the runtime performance (including execution time and memory consumption) of the centralized applications transformed by two proposed algorithms and the corresponding one without centralization.

The networked applications used in this experiment are summarized as follows :

- The Echo server sends all input back to client. The Echo client is a test client that connects to the server and sends predefined text to it (RFC 862).
- The Daytime Sever returns the current time back. The Daytime client requests the current time from the server (RFC 867).

- The Chat server returns the input of one client to all connected clients. The chat client is a test client that connects to the server, sends predefined text to it, and disconnects after having received a certain number of lines.

- The alphabet server returns the nth letter of alphabet, and the client sends fixed requests.

All experiments of each setting are repeated 50 times (including both centralization transformation and execution of the centralized application), and the averaged results are summarized in Table 3.1. The column *Bytec. Size* lists project size of each component application in a benchmark in bytecodes. Consider the Echo Server Client benchmark as an example. The size of its server is 2.17KB, and the size of each client version is 1.49KB. For each benchmark, we have four settings with different number of instances for each version of a client. The total benchmark size can therefore be calculated by taking size summation for all its projects. The transformation time and memory of cost for each algorithm are shown in columns *Trans. Time* and *Trans. Mem.*. We find that the greedy centralization takes more time and memory to perform transformation. The execution time and memory cost of the centralized application are shown in columns *Exec. Time* and *Exec. Mem.*. The execution time is the averaged real time for program execution and the memory consumption is the averaged peak memory consumption of the whole VM. They are measured by using GNU *time* tool. The storage refers to the memory cost (in KB) to store all class files of the centralized application. The execution time of the centralized applications does not show the significant difference for both transformation algorithms. The execution memory cost of the simple algorithm is slightly larger than the greedy solution because it produces more classes, which need to be loaded in to VM during runtime. The execution time and memory of the original program are larger than for their centralized counterpart. Without centralization, each application runs on its own VM which produces additional overhead. Each VM also loads its own version of classes, many of which are duplicated among different applications. Therefore, the runtime and memory consumption grows linearly with the number of applications.

Table 3.2: Application of centralization to JPF

App.	Bytec. Size [KB]	JPF.		Opt. Centra.	
		Time [h:mm:ss]	Mem. [MB]	Time [s]	Mem. [MB]
Echo Server Client	5.15	00:00:07	328	0.40	36.92
Chat Server Client	7.94	01:10:19	471	0.41	37.66
Chat Server Client v.1	7.94	00:00:01	82	0.42	37.95
Daytime Server Client	3.95	00:00:58	343	0.40	36.76
Daytime LeapSecond	6.13	00:00:14	366	0.41	37.00
Alphabet Server Client	10.11	N.A.	N.A.	0.43	39.20
Alphabet Server Client v.1	10.11	05:29:58	1023	0.42	39.28

3.4 Centralization with JPF

To show the usefulness of our centralization tool in verifying distributed application, we perform experiment on Echo client/server, Daytime client/server, Chat Server and Alphabet client/server [6] as the test beds. Each benchmark consists of one server and two clients with different versions. The verification results of JPF on the centralized application is summarized in Table 3.2. Column *Bytec. Size* shows the total size (including the server and two clients) of each benchmark, which can be calculated by summing up the size of each its application presented in Table 3.1. For a small application like the Echo client/server system, JPF finishes its verification in 7 seconds. Similarly, it takes about 1 minute for the Daytime case. The other applications are much more complex. The chat server features a high degree of internal concurrency because each request is sent back to all currently connected clients. The state space therefore contains all possibilities for concurrent client connections, with all possible permutations for establishing a connection, and for each message to be interleaved with other messages. Because of this, it takes more than one hour to finish searching the state space. The state space of Alphabet is even larger, because each client is implemented using producer and consumer threads. For the case with two active clients, this involves three threads for each application: on the server side, the main thread and two worker threads are used,

and each client uses a main thread, a producer, and a consumer thread. Because the state space is exponential in the number of threads, this case is too large for JPF to handle: After 14 hours, it reports that it has run out of memory (given 1 GB of heap space). However, while full verification is out of reach for such applications, finding defects is still possible.

We cover that use case by seeding some faults into these benchmarks. Chat. v.1 is the buggy version that has a race condition on a shared array field that stores active connections. In the failure scenario, one client disconnects, causing the server worker thread handling that client to remove that entry. Because the "remove" operation is not synchronized, another worker thread (serving a different client) checks the contents of that field (which is non-null at first) before using it. Between the check and use, the unsynchronized remove operation sets the field to null, causing the NullPointerException in the other worker thread later. In Daytime LeapSecond, the server produces a time with leap second with low probability, and one client checks the format of the time it receives. The client crashes if the time format is incorrect. These two bugs could be found by JPF quickly. However, when a large number of threads is involved, it may take more time to find a defect in a large state space. The benchmarks of Alphabet. v.1 consists more than then threads (including a wrapper thread), and it takes more than 5 hours to find the seeded bug. Previous work [5] does not support centralizing distributed applications with multiple versions. The net-iocache approach [6] analyzes each peer separately, which cannot find these bugs, either. By using our centralization approach, we can successfully find these described bugs.

Chapter 4

Program Analysis Enhanced Automatic Testing and Testing Multiple Versions

Although verification can exhaustive explore all program state such as concurrency state overlapping between different processes, it limits to medium or small size application by its state explosion problem. Therefore, software testing is still widely used and is one of the most important tasks during software development to improve the reliability and correctness of software system. Compared to verification, testing techniques are more robust and scalable, while verification is more widely used for concurrent systems, resulting both approaches has their own application scenario. Continued last section, this section proposes and discusses our techniques in testing multiple version variants, sharing testing results among multiple versions while reduce the redundancies in testing the common code. We first discuss our program analysis enhanced random testing techniques and then discuss our refined project centralization based techniques for testing multiple versions.

With the increasing application of revision control system like SVN, Github, Mercurial, and Software Product Lines, more and more similar product variants are created, where these products share some common code and features. Testing these similar product variants separately, however, causes lots of redundancies in testing their common code, which may loose the chance to testing their variabil-

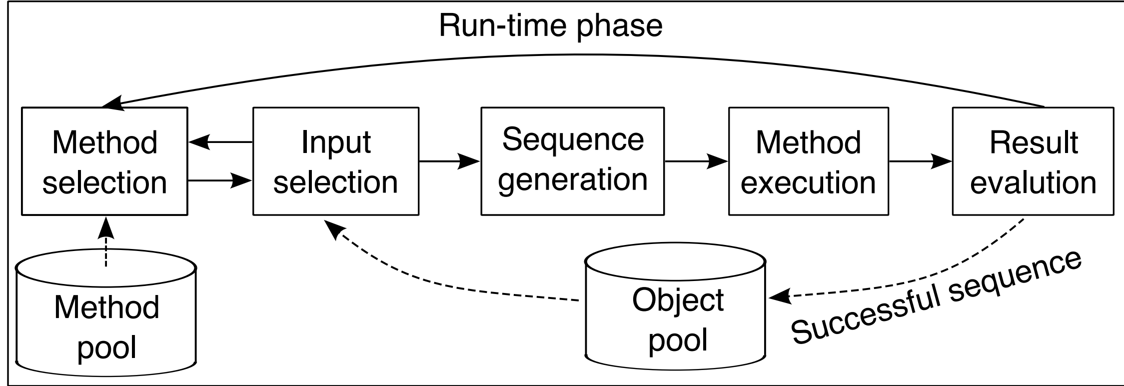


Figure 4.1: Flow of default Randoop.

ity under the cost constraint. To the best of our knowledge, existing automatic software testing and test case generation techniques can only test one product at a time. Although they could be applied testing each of the similar products separately, they cause redundancies in code representation and test case execution. They cannot share the testing results to further improve the testing performance of another products, either. When testing multiple products, we may generate test cases towards multiple optimization goals, like average code coverage, test case length, memory consumptions, and so on.

4.1 Program analysis enhanced random testing

4.1.1 Introduction and Motivation

Manually crafting test sequences is a labor-intensive task. Random testing automatically generates test sequences to execute different paths in a method under test (MUT) [34]. It randomly constructs object instances as the receiver and input arguments of the MUT. However, we found that existing random techniques suffer from low code coverage. Reasons are that randomly generated sequences may not able to set up the receiver in all the required states, or that the required input arguments for invoking the MUT cannot be generated automatically.

Feedback directed Random testing approaches and its tool Randoop represent the state of the art in automatic random testing. Randoop [66] implements a random testing approach for method sequence generation. Given the software to test, Randoop first extracts all publicly visible API methods and puts them into a

fixed *method pool,* which contains all methods to be considered for testing. Randoop also includes simple primitive values and a few simple objects such as strings, in its initial *object pool* (see Fig. 4.11).

Randoop tests MUTs by randomly selecting a method $m(T_1 \dots T_k)$ to test from its method pool. All input objects with type $T_1 \dots T_k$ must also be prepared to test m. Randoop randomly selects the corresponding inputs from its object pool and concatenates previously known input sequences to derive these known objects, to test m. If there exists no object with the required type in the object pool, Randoop skips m and selects the next method. Upon successful input construction, m is executed. Execution results of each method call are analyzed against a few predefined contracts. If the generated sequence is new and its execution does not cause any failures, it adds these *successful sequences* to the object pool (see Fig. 4.11).[1] Randoop continues to test more methods until a time limit is hit.

We found that Randoop shows several limitations in practical applications despite its versatility:

1. Small constant pool. Randoop uses a small pool of predefined constants and primitive values observed at run-time as inputs. Many relevant values are missed.

2. No distinction between methods with and without side effects. Pure methods do not change the object state and may add long redundant subsequences to a test.

3. Static type management. This limits coverage in cases where the dynamic type differs from the declared type.

4. Fixed method pool. An unordered, fixed set of methods to test may prevent some methods from ever being executed due to unfulfilled dependencies.

5. Lengthy input sequences. Randoop has no bias towards light-weight methods, which in general results in long test sequence execution times at a later phase.

6. No run-time coverage guidance. Randoop selects the methods to test with no bias, choosing easily covered and hard-to-cover methods with the same probability.

[1] It is assumed that execution sequences are deterministic.

```java
package org.apache.commons.cli;
public class PatternOptionBuilder{
  public static final Class STRING_VAL=String.class;
  public static final Class OBJECT_VAL=Object.class;
  // 7 more similar fields omitted.
  public static Object getValueClass(char ch) {
    switch (ch) {
     case '@':return PatternOptionBuilder.OBJECT_VAL;
     case ':':return PatternOptionBuilder.STRING_VAL;
     // 7 more case branches omitted.
    }
    return null;
} } // 2 more methods omitted.
public class TypeHandler {
  // 1 method omitted.
  public static Object createVal(String s, Class c) {
    if (PatternOptionBuilder.STRING_VAL == c)
      return s;
    else if (PatternOptionBuilder.OBJECT_VAL == c)
      return createObject(s);
    // 7 more else if branches omitted.
    else return null; } } // 7 more methods omitted.
```

Figure 4.2: Two classes from *Apache cli*. Branch coverage requires both domain knowledge on constant values and accurate type management.

To address these limitations, we first perform a static analysis on the classes under test. This step extracts domain knowledge such as possible constants during execution, method side effects, and their dependencies. The results are later combined with the run-time input, demand-driven object construction, and coverage information techniques, to guide testing to those MUTs with low coverage. We manage the generated sequences based on the dynamic type information and favor sequences with low execution time as input to other MUTs. Our approach is fully automated and requires no specification of possible inputs. We implement our techniques as pluggable options based on Randoop. The evaluation of our approach on 30 popular real-world applications demonstrates its effectiveness.

4.1.2 Weakness Found in Randoop

Existing random testing techniques suffer from low structural coverage on practical programs, where many diverse input object states are required to cover all code. Although formal specifications with well-defined input and method invocation con-

```
1  package org.apache.commons.compress.utils;
2  public final class IOUtils {
3    private IOUtils() { }
4    public static long copy(final InputStream input,
5      final OutputStream output) throws IOException {
6      return copy(input, output, 8024); }
7    public static long skip(InputStream input, long n)
8      long available = n;
9      while (n > 0) {
10       long skipped = input.skip(n);
11       if (skipped == 0) break;
12       n -= skipped;
13     }
14     return available - n; } } // 5 methods omitted.
```

Figure 4.3: Methods in *Apache compress* that require inputs outside the fixed method pool of Randoop.

straints enhance random testing by shrinking the possible input space, they are often unavailable or incomplete in practice. This limits their applicability.

Randoop is fully automated, using no specification or model. This makes Randoop very easy to use; however, there are still areas in which code coverage is less than optimal.

4.1.2.1 Small Initial Value Pool

Randoop stores a small set of simple constants for primitive types in the initial prefixed value pool, such as -1, 0, 1, 10, 100 for bytes, or "hi!" and "" for strings. It incrementally grows its initial object pool by storing objects derived from executed test sequences. However, the lack of diversity in the beginning causes it to miss many branch conditions. Furthermore, Randoop cannot observe primitive temporary values that are used inside a method. Therefore, even when including all initial values and the values obtained at run-time, the small value pool still limits the achievable coverage. Consider the example from class `PatternOptionBuilder` in Fig. 4.2: Branches of method `getValueClass` are not covered by Randoop as it does not contain predefined primitive values, or values derived from them, to satisfy these branch conditions at run-time. However, using primitive values such as '@' as an input would easily cover a branch in this case.

4.1.2.2 No Distinction on Side Effects

Only methods with side effects can change the state of an object [83]. These methods should be favored over methods without side effects in order to frequently mutate object states and, thus, to satisfy more branch conditions. Since Randoop does not distinguish between methods with and without side effects, it creates long and redundant sequences for objects staying in the same state. All of these sequences are put into the pool of reusable sequences, deferring state changes even further and slowing down coverage growth.

4.1.2.3 Static Type Management

Randoop uses the static (declared) return type of a method to manage a successful sequence in the object pool. Static type management does not distinguish whether a method produces the declared return type or a subtype at run-time. It therefore may not cover all possible behaviors of a method, causing some methods never to be tested because no compatible input seems available, even if a sequence that returns the required type at run-time does exist. This limitation may result from the need to easily generate compilable code for off-line testing, which requires the correct type casts to be added if the dynamic type of a variable does not correspond to its static type.

In our example in Fig. 4.2, branch coverage in method `createVal` requires both suitable primitive values and a class descriptor returned by `getValueClass` method. However, static type management stores the object returned by `getValueClass` as type `Object` according to its declaration. This results in `createVal` never being directly tested.

4.1.2.4 Fixed Method Pool

Randoop uses a fixed method pool when considering methods to test. However, many methods require an input type that cannot be generated from invoking a fixed set of API methods; instead, they may require data from libraries not belonging to the software under test itself. When a method requires an input type that cannot be generated from existing input sequences, Randoop simply skips that method. Because of this, many MUTs are never tested.

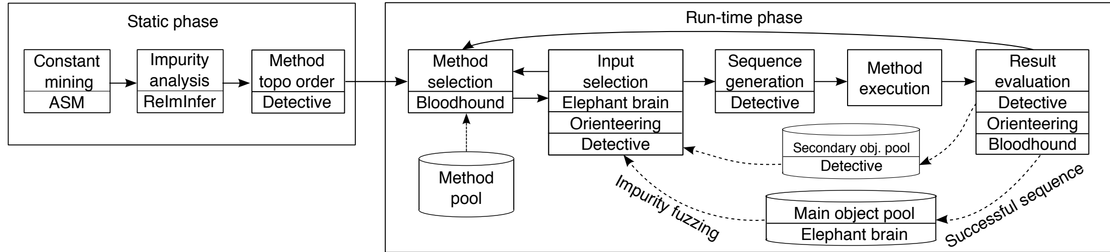


Figure 4.4: Flow of our enhanced Randoop.

Consider class `IOUtils` (see Fig. 4.3), where both methods `copy` and `skip` require an object of type `InputStream`. However, the required object is never generated by Randoop as it requires using the Java core library. Due to this, no method of this class is ever covered, even though providing the required input easily covers most of these methods.

4.1.2.5 Lengthy Input Sequences

Randoop manages all generated successful sequences in its object pool such that sequences that return the same type are put into the same set. Whenever an input with a specific type is required, Randoop randomly selects a sequence among all available sequences. Since new sequences are constructed by concatenating existing input sequences, the resulting sequences can grow considerably in length. This adds to the execution cost of generated test cases. Randoop quickly reaches a bottleneck after running for several minutes, repeatedly executing lengthy sequences while leaving many other relevant sequence combinations untested.

4.1.2.6 No Coverage-Based Guidance

The difficulty of covering a branch varies between branches. Some branches are easily covered in the early phase of random testing, while others require a complex object state. An equally balanced selection of methods to be tested, wastes time on those methods that are already well covered. On the other hand, too much emphasis on uncovered branches may waste time in challenging the difficult branches without much payoff. Our observation is that the current strategy of Randoop is not ideal, but the solution is not as simple as looking for uncovered branches [44].

4.1.3 Our Enhancements

We have devised six enhancements to Randoop. They are based on information gathered from a static analysis prior to running the main tool, and at run-time when generating tests (see Fig. 4.4). The information is used to guide Randoop. We briefly define a term to denote each enhancement and describe them in detail below.

1. *Constant mining* (static): Constants are extracted from the classes under test to seed the initial value pool.
2. *Impurity* (static + run-time): Inputs are fuzzed based on Gaussian distribution and a method purity analysis.
3. *Elephant brain* (run-time): Input sequences are managed with the exact return type obtained at run-time.
4. *Detective* (static + run-time): We test methods in topological order and construct missing input data on demand.
5. *Orienteering* (run-time): Method sequences that require less execution time are favored.
6. *Bloodhound* (run-time): Method sequence generation is guided by coverage.

4.1.3.1 Automatic Constant Extraction

As observed earlier, Randoop misses many branches because its initial pool of input values is limited. Various techniques exist to remedy this shortcoming. Symbolic execution [56, 95] and concolic testing [31, 76] use symbolic constraints to derive inputs that cover more branches. Unfortunately, symbolic execution is slow and does not deal well with complex branch conditions and nested data structures.

Adaptive Random Testing (ART) [16, 19] searches for possible values from the whole input domain based on distance and a given distribution. However, the input domain space is usually very large, which limits ART in covering even simple branch conditions such as a string comparison.

To obtain relevant input values without incurring much overhead, we perform a lightweight static analysis on the classes under test, which we call *constant mining*.

Our analysis extracts constants, and it performs constant propagation and constant folding to further extend the set of input values. The extracted values are fed into the value pool with the assumption that many of these values are close to satisfying some branch conditions.

For example, branch conditions such as the string comparison `str.equals("coverMe")` can be easily covered. To obtain even more possibly relevant values, state fuzzing techniques (see Section 4.1.3.2) are used.

Our tool for extracting constants is based on ASM [12]. We implement our abstract interpreter on top of the ASM interpreter framework, which analyzes the Java bytecode of each class under test and performs constant propagation and folding. Our analysis is intra-procedural; we forgo a more accurate inter-procedural and pointer analysis for performance reasons. Our approach goes beyond value extraction from source code [72].

4.1.3.2 Purity-Based Object State Fuzzing

The coverage of a specific execution path requires the receiver as well as the input argument objects to be in a certain state. In order to generate sequences with a broad variety of object states, we alter *(fuzz)* the states of the input objects and pass the fuzzed results to the MUT. We handle primitive values based on a Gaussian distribution and non-primitive objects based on method purity analysis.

4.1.3.2.1 Primitive Object Fuzzing Primitive inputs are either extracted by constant mining or from run-time execution results. To cover a wider range of inputs, we use a heuristic that assumes that given values are already close to satisfying some of the branch conditions. We adopt a Gaussian distribution to probabilistically select the next value as input, where we use the originally selected input value and a prefixed constant as the expectation (mean) value μ and deviation σ, respectively. This approach gives higher probability to values close to μ (68.3 % of all values probabilistically lie in $[\mu - \sigma, \mu + \sigma]$), while still generating also some values far from μ.

To prevent newly generated primitive values from polluting our object pool, we do not store values obtained from fuzzing; we only add new values that are observed during execution. This allows us to obtain a wider range of primitive values based on information extracted from the classes under test. We implement our primitive

value fuzzer based on the elementary distance [19] to generate values. In our current setting we fixed $\sigma = 30$. The Levenshtein distance [54] is adopted for generating string values.

4.1.3.2.2 Purity-Analysis-Based Object State Fuzzing In order to increase the coverage of branches and execution paths, the obtained objects have to be in diverse states. To fuzz non-primitive objects, we identify those methods that mutate the object state by having a side effect. Side effects may affect either the receiver or input arguments (parameters).

Method purity analysis [83] classifies MUTs into pure and impure methods. Methods without side effects are *pure*, methods with side effects are impure. We focus on method *impurity* for fuzzing input object states.

Static [38, 83, 91] and dynamic [92, 98] purity analysis techniques and tools exist. We choose a static technique to avoid any overhead during the run-time phase. We implement our method purity analysis based on the ReIm & ReImInfer framework [38] due to its scalability and robustness. This framework does not require an expensive, complete analysis of the program. It infers method purity based on the type system and by its automatic inference algorithm. Since our impurity enhancement is probabilistic, a fast but sometimes inaccurate purity analysis fulfills our need.

4.1.3.3 Dynamic Input Sequence Management

To improve the accuracy of input object selection, we manage the object pool by using the exact return type of each method sequence, obtained at run-time. Our approach easily covers previously uncovered methods that depend on the exact type of its input (due to the subtype relation, in cases where the exact type is cast to its super class and not remembered by default Randoop).

When outputting the generated sequences as JUnit test cases, we also compare the static type of each method to its dynamic type, adding explicit type casts where needed. Without type casts, the generated JUnit code would fail to compile because the static type of a variable or return value does not match the dynamic type requirement of its receiver.

This enhancement generates many different data types, and never forgets; we therefore call it *elephant brain*.

4.1.3.4 Demand-Driven Input Construction

To test an MUT, its receiver object and input arguments must be prepared from the object pool. Default Randoop skips a method if some input type is unavailable.

Randoop only considers publicly visible methods and constructors from the classes under test for testing. However, this misses cases where a method of a different class or another library returns the right type.

Although it may be tempting to put additional methods from other classes into the method pool, this greatly increases the search space and pollutes the object pool, wasting effort on methods that are not the target.

We make two improvements: First, we sort MUTs by their topological dependency order. Second, we use a demand-driven approach to construct inputs that are not directly available by leveraging a secondary object pool.

Our topological-order analysis statically computes dependencies of MUTs by considering their input and return types. Given two methods A and B, A depends on B if A requires an input type that can be returned by B. We compute this dependency information as a graph, and sort the methods in topological order so that methods with no input dependencies execute before methods that need input returned by other methods. After executing all methods in topological order, we can quickly identify those methods that lack input for testing, and only consider those unavailable inputs as candidates for demand-driven input construction.

Our demand-driven approach searches all available packages for constructors and static methods that return a required type. We recursively search further, to a maximum of five nesting levels, if more input is needed for a candidate method that returns the sought-after type.

If a set of methods to produce the required data is found, we execute the constructors and methods in topological order and store all obtained objects in a *secondary object pool.* The secondary object pool is only used for previously unavailable input types. A sequence that directly returns a previously required input type is added to the main object pool, while other generated sequences remain in to secondary object pool. If further input is needed later, we first check the secondary object pool before performing an expensive recursive search again. To reduce the overhead of the secondary object pool, we keep only one sequence for each type. Our approach does not pollute the main object pool and provides more diverse input types with little overhead.

Like a *detective*, this enhancement often uncovers new relationships between methods.

4.1.3.5 Cost-Guided Input Sequence Selection

Randoop manages generated method sequences by their return types. All sequences that return the same type are treated equally, regardless of their length and execution time.

For better run-time performance, it is desirable to use method sequences that require less execution time. The idea is inspired from *orienteering*, where a path that takes less time is preferable. Therefore, we select a sequence based on its execution time: $\text{weight}(seq) = 1/(seq.exec_time * k)$, where *seq* is a sequence for selection, k counts how many times *seq* has been selected so far, and *seq.exec_time* is the execution time of *seq*. As measuring and updating the execution time of each sequence incurs additional overhead, we measure the execution time of each sequence only once, expecting it not to change much.

4.1.3.6 Coverage-Guided Method Selection

To direct Randoop towards uncovered code, we perform a coverage analysis during test generation, and favor those MUTs that are not well covered. Although it is desirable to update the coverage information after executing each MUT, this is expensive; method selection in Randoop is executed very often. Therefore, the coverage information is updated with time interval t. During each interval, we prioritize method selection for a method m among all MUTs M by using the following function as its weight $w(m, k)$:

$$\begin{cases} \alpha * \text{UncovRatio}(m) + \beta * \left(1 - \dfrac{\text{Succ}(m)}{\text{MaxSucc}(M)}\right) & \text{if } k = 0 \\ \left(\dfrac{-3}{\ln(1-p)} * \dfrac{p^k}{k}\right) * \gamma * w(m,0) + \dfrac{\delta}{\text{Size}(M)} & \text{if } k \geq 1 \end{cases}$$

In this function, k represents the number of selections of method m since coverage information was last updated; $\text{UncovRatio}(m)$ is the uncovered branch ratio of m; p is the parameter of a logarithmic series that determines how fast the factor decreases as k increases; $\text{Succ}(m)$ is the total number of successful invocations of m;

MaxSucc(M) is the maximal number of successful invocations of all MUTs; Size(M) is the number of MUTs M; and $\alpha, \beta, \gamma, \delta$ are parameters to adjust the weight of each formula.

The overall effect of the weight function is that initially ($k = 0$) we favor those methods with low code coverage. Once a method has been tested successfully ($k \geq 1$), we downgrade its weight logarithmically. After several rounds of selection, the weights return to a nearly-uniform distribution again. At each update of the coverage information, the weights are recalculated, and k is reset to 0.

Our method selection strategy is inspired by the multi-armed bandit algorithm [93]. This algorithm balances "exploitation" (methods that are well tested) and "exploration" (methods with low coverage) for a higher payoff. This algorithm is useful because some branches of a MUT can be difficult to cover even if a method is tested often. A weight function only based on an uncovered ratio of code would waste resources on those methods with difficult branch conditions without gaining much benefit. Our approach considers both code coverage and the execution history of each MUT for the initial weight, but decreases this weight later to avoid wasting too much effort in difficult branches.

We implement this enhancement using JaCoCo [39]. The original version of JaCoCo provides only off-line coverage analysis based on Java bytecode. For our work, we enhance JaCoCo to enable on-line profiling using dynamic instrumentation. During class loading, we insert probes before each branch or jump instruction, so that if the corresponding part of the code is executed, the probe flag will be set. Coverage is periodically updated based on this information. Currently, we set the weight parameters to $p = 0.99, \alpha = 0.3, \beta = 0.7, \gamma = 0.9, \delta = 0.1$; we leave parameter tuning as future work.

Like a *Bloodhound,* this enhancement hungers for coverage, while intelligently balancing the deeper search of each MUT against the breadth given by the entire problem set.

4.1.4 Experiments

We evaluate enhancements to Randoop by investigating the following questions:

1. How much does coverage improve compared to original Randoop?

2. How much does each enhancement improve coverage?

3. Does coverage reach a saturation point more quickly or slowly with the enhancements?

4. How many defects can be found in real software thanks to the enhanced version?

5. How many false positives (incorrect API usage and the like) are caused by our enhancements?

4.1.4.1 Setting and Methodology

We evaluate our enhancements in comparison to the original version of Randoop 1.3.4 (released in January 2014). We use Randoop with default settings, unless indicated otherwise. Each of our enhancements can be activated separately, allowing us to evaluate them individually.

As basis for our evaluation, we select a collection of 30 popular software packages. The overview in Table 4.3 shows for each package its name and version, the number of classes it contains, and the number of methods. From these classes and methods, only a subset constitutes the public API. These are listed as "public" and contain public classes and methods as declared in the source code. We also indicate the number of branches in all methods, their total cyclomatic complexity (the number of linearly independent paths) [60], as well as the overall size of the program in terms of non-comment lines of code (NCLOC).[2]

All experiments are run on an Intel Core i7 Mac 2.4 GHz with 8 GB of RAM, running Mac OS X 10.9.1 and Oracle's Java VM (JVM), version 1.7.0_21. We use a memory limit of 3 GB for the JVM, except for large benchmarks where 4 GB is needed.[3]

4.1.4.2 Coverage Evaluation

We evaluate coverage improvements on 30 selected software packages and investigate the behavior of our enhancements on the SCCH collection library [71] in detail.

[2]Using CLOC 1.60, `http://cloc.sourceforge.net/`.

[3]4 GB is used for Apache commons math and compress, ASM, and Guava. For Apache commons math and compress, we further turn off "constant mining" when combining all enhancements, because these packages contain so many constants that the memory is quickly exhausted. Finally, WheelWebTool causes problems with the class loader, forcing us to turn off the Detective enhancement.

Table 4.1: Benchmarks: Size and complexity metrics, and test results

Software	# classes		# methods		#	Cyclom.	NCLOC	Coverage (default)			Coverage (enhanced)		
(version)	public	all	public	all	branches	compl.		# m.	ins. [%]	br. [%]	# m.	ins. [%]	br. [%]
A4J (1.0b)	45	45	518	522	544	794	3,602	519	81.5	48.7	521	84.0	54.0
Apache BCEL (5.2)	333	383	2,679	3,182	5,227	5,613	23,631	1,206	32.3	19.3	1,615	44.7	30.8
Apache Commons Cli (1.2)	18	22	162	211	490	455	1,978	191	72.7	55.1	193	76.4	64.1
Apache C. Codec (1.9)	58	85	441	625	1,835	1,539	5,803	513	87.1	69.5	572	93.7	81.4
Apache C. Collection (4.0)	312	431	2,866	3,861	5,499	6,225	23,713	1,853	38.6	28.1	2,782	65.9	52.8
Apache C. Compress (1.8)	111	191	1,066	1,691	4,634	3,943	17,462	645	17.5	14.7	1,078	53.0	36.4
Apache C. Math (3.2)	736	1,001	6,485	8,089	18,576	16,255	81,792	3,504	38.5	23.9	4,773	55.2	35.3
Apache C. Primitive (1.0)	154	259	1,826	2,243	1,446	2,581	9,836	1,032	57.1	65.2	1,100	62.0	70.8
ASM (5.0.1)	98	166	1,402	1,877	7,050	5,622	24,193	1,072	35.5	23.5	1,256	43.6	29.9
Beanbin (1.0b)	88	90	361	482	666	779	3,786	229	26.4	23.1	243	31.6	27.3
ClassViewer (5.0.5b)	7	24	88	147	470	356	1,485	64	36.1	19.1	79	48.2	36.6
Dcparseargs (R4)	6	6	22	22	88	65	204	21	51.4	58.0	21	54.1	63.6
Fixsuite (R48)	25	42	196	275	322	428	2,665	122	19.5	15.5	124	22.1	19.3
Follow (1.7.4)	65	96	331	460	487	682	4,812	100	16.8	21.4	158	26.5	31.6
Guava (16.0.1)	365	1,678	7,989	12,928	11,247	16,214	66,566	3,964	34.0	26.0	4,920	44.1	37.1
Java Simp. Arg. Parser (2.1)	59	69	418	508	714	849	4,888	387	60.0	50.6	410	70.2	62.5
Java View Control (1.1)	24	25	153	268	2,064	1,466	4,617	57	6.2	4.6	146	31.8	13.2
Javax Mail (1.5.1)	229	311	1,831	2,692	9,523	7,316	28,271	1,379	35.4	25.8	1,418	37.3	27.1
Jdom (1.0)	53	75	737	925	3,196	2,467	8,362	648	45.7	31.0	715	56.6	39.7
Joda Time (2.3)	144	232	3,460	4,314	6,172	7,049	27,638	2,765	62.1	45.1	3,271	73.6	55.2
Jsecurity (0.9.0 RC3)	133	136	667	797	704	1,038	13,135	485	40.6	16.8	503	46.1	23.3
Lotus (R29)	55	57	126	138	131	193	1,028	100	59.0	30.5	106	63.1	36.6
Mango (2.1 03/2014)	78	97	354	421	382	598	2,141	299	70.2	55.2	345	81.6	70.2
Nekomud (R16)	10	11	33	37	44	53	363	7	7.9	9.1	9	10.8	13.6
Rhino (1.7R1)	122	259	1,534	3,583	19,422	13,938	43,178	1,282	18.9	12.8	1,531	24.6	18.2
SAT4J Core (2.3.5)	204	265	2,208	2,690	3,815	4,235	17,397	1,636	43.6	25.6	2,015	64.0	43.3
SCCH collection (1.0)	23	37	198	261	308	339	1,348	133	59.1	44.5	136	67.5	57.8
Tiny Sql (2.26)	29	31	657	719	2,237	1,825	7,672	449	25.3	17.1	486	32.1	23.4
TullibeeAPI (R64)	21	21	207	242	909	701	3,236	138	30.2	14.0	139	30.3	14.5
WheelWebTool (0.8.2a)	108	122	1,277	1,517	4,988	4,006	17,581	582	23.4	19.4	585	24.8	21.8
Total (average for coverage)	3,713	6,267	40,292	55,727	113,242	107,624	452,383	63.0 %	41.1	30.4	77.6 %	50.7	39.7

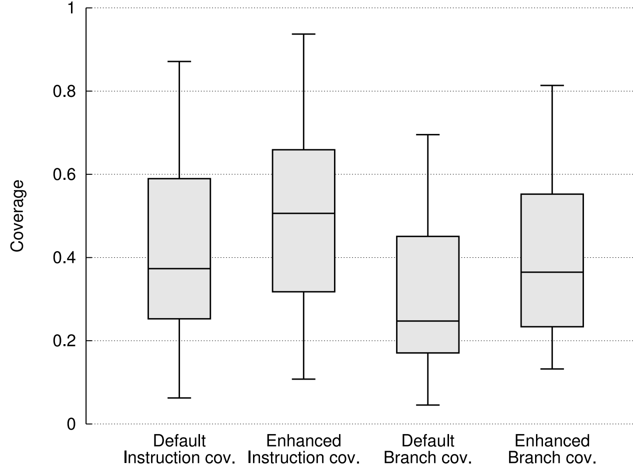


Figure 4.5: Instruction and branch coverage of default and enhanced Randoop (after 3000 s).

4.1.4.2.1 Open Source Benchmarks Coverage data in terms of method, instruction, and branch coverage is summarized in Table 4.3 for all packages, where each configuration runs with 3000 seconds as time limit.

Fig. 4.5 shows the overall coverage improvement graphically, as a box plot. The plot depicts the data as quartiles, i. e., the box contains the spread of 50 % of all cases. The horizontal line inside the box indicates the median. The whiskers at the top and bottom show the extreme cases. Our enhancements show a significant improvement of coverage for all cases. For instruction coverage, we obtain $\mu = 41.09$, $\sigma = 21.16$ for default Randoop, over which our enhancements show a significant improvement: $\mu = 50.67$, $\sigma = 20.96$ at $p < 0.05$ (Wilcoxon Matched-Pairs Signed-Ranks Test). For branch coverage, the values are $\mu = 30.44$, $\sigma = 17.69$ (default Randoop), $\mu = 39.71$, $\sigma = 19.16$ (enhanced Randoop); $p < 0.05$ (Wilcoxon Matched-Pairs Signed-Ranks Test).

2 In general, each individual enhancement shows some improvement; the impact of each enhancement varies across different cases. The combination of all enhancements is much stronger than each enhancement by itself, which can be seen for instruction coverage but is even more apparent for branch coverage (see Fig. 4.6). However, in some cases, certain enhancements worsen performance (see Fig. 4.9). Fine-tuning the parameters to counteract this is future work.

3 Our enhancements provide much faster coverage saturation but still allow for some incremental improvements when default Randoop already tapers off (see Fig. 4.7–4.9). Usually the Orienteering enhancement is effective by itself, but there

are also cases where the second phase of the detective enhancement makes a huge difference (see Fig. 4.8).

To save space, we put other plots online at `https://staff.aist.go.jp/c.artho/ase-2014/`.

4.1.4.2.2 SCCH Collection The SCCH collection package is based on a reimplementation of common Java collection classes (such as list, array, set, stack, and map) for teaching purposes. It has been used in our previous experiments on unit testing and test case generation [71], where we manually seeded defects. We therefore chose this package to investigate more closely how coverage is affected by the different enhancements.

The behavior of collection classes is heavily influenced by their internal state in terms of the number of elements they contain. Since the elements are only stored and not processed, their value is usually not of importance unless it is `null` or the collections are ordered. Thus, for this type of software, operations that mutate the object state help to increase the coverage and to trigger failure cases.

Consequently, as one can see in Fig. 4.9, the Impurity enhancement improves coverage results. However, Orienteering does even better because some operations that mutate the state are also fast and thus frequently used by this heuristic. Constant mining provided elements of different types, which increases the coverage in cases where elements are compared. In contrast, the Bloodhound tends to favor diverse sequences, which limits the chance for state changes. Selecting methods based on the coverage works poorly in case of collection classes. While the two-phase approach of Bloodhound mitigates this problem to a large extent, it does not produce a coverage improvement in this case.

In line with our previous case studies [71], we also run all the enhancements in varying configurations and including the option of using `null` as input value at a probability of 10 %. The results are summarized in Fig. 4.10. They show that the enhancements can have a mixed, positive and negative effect. Furthermore, using `null` values also has a major impact on how our enhancements interact. Although we get a similar overall improvement, the Impurity enhancement is much less useful in combination with `null` inputs than under default settings (see Fig. 4.9).

We are aware that container classes represent a special case. However, our detailed observations on the SCCH collection package show that achieving a good

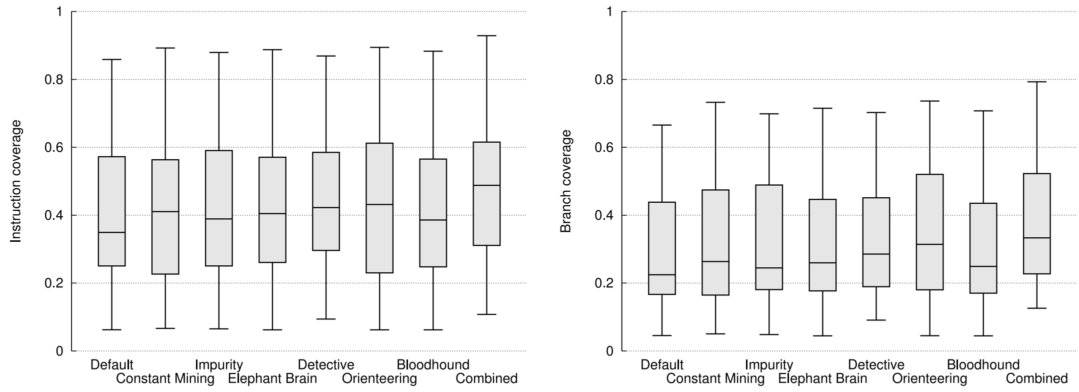


Figure 4.6: Instruction and branch coverage for each enhancement (after 600 s).

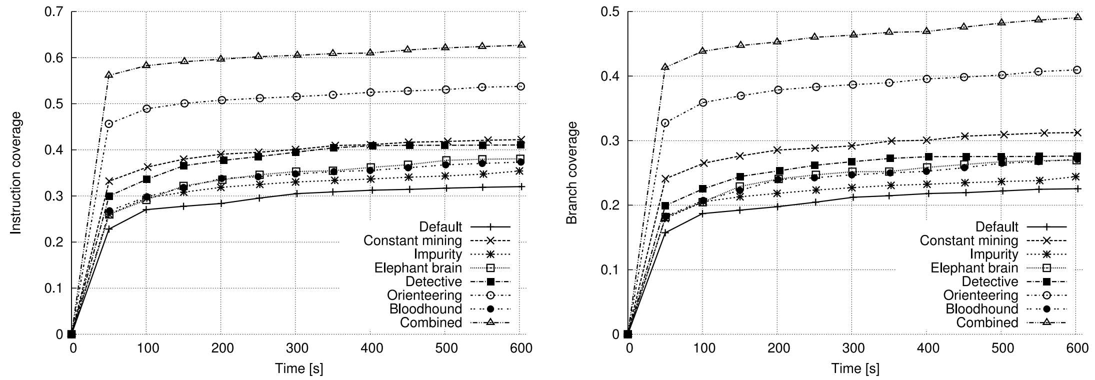


Figure 4.7: Coverage improvement over time (Apache commons collections).

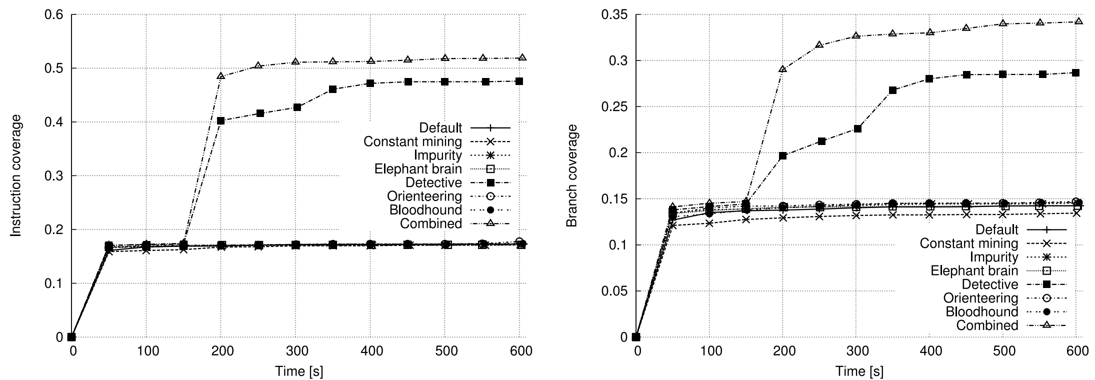


Figure 4.8: Coverage improvement over time (Apache commons compress).

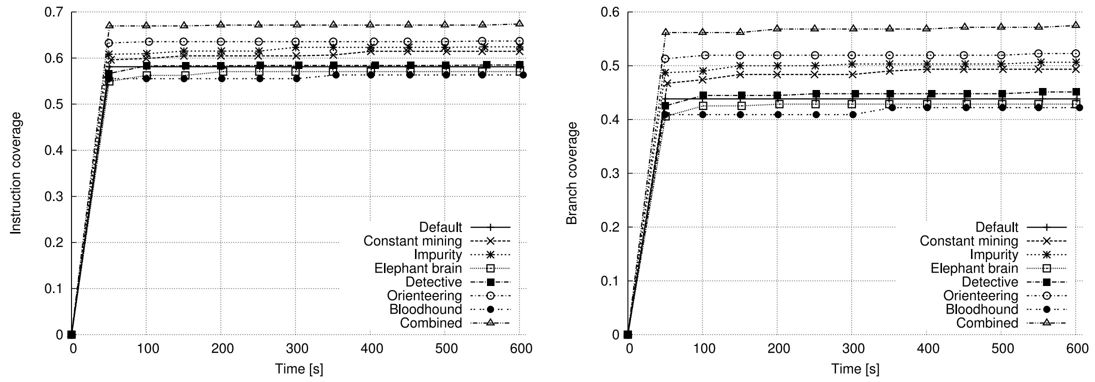


Figure 4.9: Coverage improvement over time (SCCH collections).

		Constant mining	Impurity	Elephant Brain	Detective	Orienteering	Bloodhound	Combined
AbstractCollection	68%		-0%					
ArrayList	96%	-0%	+3%	+1%	-1%	+3%	+0%	+3%
ArrayList.ArrayListIterator	42%							
Arrays	54%	+6%	-22%	+20%	-2%	+40%	-9%	+39%
Collections	91%	+0%	-6%	+5%	-10%	+9%	-2%	+9%
Collections.DefaultComparator	100%							
Collections.ReverseComp.	50%	+20%	-5%	+20%	-10%	+50%		+50%
ConcurrentModificationExc.	100%							
EmptyStackException	100%							
HashMap	40%							
HashMap.Pair	0%							
HashSet	73%	+1%	+3%	+3%	-3%	+4%	-3%	+5%
HashSet.HashEntry	64%							
HashSet.HashSetIterator	56%							
LinkedList	97%	-0%	+0%	+1%	-1%	+1%	-0%	+1%
LinkedList.LinkedListIterator	36%							
LinkedList.Node	100%							
MapImpl	14%							
NoSuchElementException	100%							
Stack	92%							
TreeMap	56%							
TreeMap.Pair	0%							
TreeSet	54%	+4%	+3%	+10%	+3%	+17%	+1%	+22%
TreeSet.AANode	100%							
TreeSet.TreeSetIterator	40%	+7%	-3%	+8%	+1%	+16%	+6%	+18%
average	65%	2%	-1%	3%	-1%	6%	0%	6%

Figure 4.10: Behavior of our enhancements on the SCCH collection classes, with `null` values enabled.

coverage throughout diverse software packages is very difficult, and that combining multiple enhancements can help to counteract weaknesses of a specific heuristic in many cases.

4.1.4.3 Defect Detection

4 We study defect detection on the SCCH classes, which are well understood [71], and on ongoing open-source projects, where the problem is much more open. In the former experiment, both Randoop and our enhancement find 9 out of the 37 known, seeded bugs in the SCCH collection. The reason why our enhancements do not find more defects is that (1) coverage increases in areas without seeded defects, and (2) most defects cannot be detected with the generic built-in test oracles of Randoop.[4]

For a more in-depth evaluation of defect detection capabilities, we apply default and enhanced Randoop to the most recent version of popular software projects, with the goal of uncovering new, previously unknown defects.

From the failed tests in the earlier evaluation, we choose projects where the number of failed test cases reported is not prohibitively high (i. e, fewer than 100 failed tests for both versions combined). From this list we choose a subset that is still under active development, with the last update on the web page or source code being less than a year ago. This selection results in ten projects (see Table 4.2).

The number of test cases to be considered is still very high at this point, with a total number of over 300 tests. We first filter out a number of tests that confirm a problem that is either known or not going to be fixed in the code. These issues include:

- **Deprecated methods.** Methods may be marked as *deprecated* if they contain or expose a design flaw. A deprecated method may allow incomplete objects to be used before all invariants are established. Such methods are usually removed from the library in the future.

  In our context, tests involving such code confirm previously known issues. These are usually true positives but as the code may be removed soon, the

[4] These built-in oracles cover behaviors stipulated for base class `Object`, which apply to all subclasses as well. More specialized checks would require in-depth knowledge of the behavior of a class, which cannot be provided automatically with current technologies.

defects may not be fixed anymore. Ignoring these issues allowed us to focus on unknown flaws.

- **Stack overflows.** Container classes, such as found in the Apache collection library or Guava, allow recursive nesting of data. For example, a list l_1 can be inserted into another list l_2, followed by an insertion of l_2 into l_1. Operations such as list iteration or `toString` will then never terminate on the resulting objects.
  It is possible to make iteration robust against infinite recursion, but a fix entails keeping track of previously visited object instances during iteration. This requires an amount of memory that is linear in the size of the collection; the cure would be worse than the disease in most cases.

- **Internal packages.** Such packages are common in Oracle's projects and usually start with `com.sun`. They occur in project `javax.mail`. Internal packages are specific to the given reference implementation and not meant to be used by others. Possibly unsafe API uses found by test cases are therefore irrelevant.

This filtering reduces the number of failed test cases found by default/enhanced Randoop to 97 and 151, respectively (see Table 4.2). We then manually simplify the remaining tests further. Often it becomes apparent that a given test is a variation of an earlier one. This can be confirmed by comparing their stack traces and the sequence of method calls. Test minimization also may confirm if two failing tests are exposing the same issue.

As this phase is still very time consuming and difficult, we limit our analysis of failed tests in *Apache math* to the first 20 tests found by each tool.[5] This reduces the test to a final number of 63 and 95 tests, respectively.

We classify these failed tests into distinct *issues*. When doing so, we count the same defect in two classes as two issues, as well as two different types of problems in the same class. This results in 31 and 49 issues found by both versions of Randoop. Each issue is then investigated more closely, and based on the API documentation and our own experience, we discount some as false positives.

[5]Skipping the remaining tests actually slightly favors default Randoop as it discovers fewer failed test cases.

Table 4.2: Defect detection capability of default and enhanced Randoop

Software	Default Randoop							Enhanced Randoop							Issue numbers
	tests	depr./ign.	issues	false	unkn.	true	only	tests	depr./ign.	issues	false	unkn.	true	only	
A. CLI	0	0	0	0	0	0	0	1	1	0	0	0	0	0	–
A. Codec	1	0	1	0	0	1	0	2	1	1	0	0	1	0	183, 184
A. Collection	27	0	8	7	0	1	0	56	25	15	13	0	2	**1**	512–516
A. Compress	7	0	1	0	0	1	0	25	0	4	0	0	4	**3**	273–276
A. Math	54	9	9	4	2	3	0	76	12	7	2	2	3	0	1115–1118
A. Primitive	0	0	0	0	0	0	0	4	0	2	1	1	0	0	17
Guava	2	0	2	1	0	1	0	13	5	8	6	0	2	**1**	1722–1724
JavaMail	17	9	5	0	0	5	0	20	12	6	0	0	6	**1**	6365–6368
Mango	0	0	0	0	0	0	0	3	1	1	0	0	1	**1**	1
TinySQL	7	0	5	1	4	0	0	8	0	6	1	5	0	0	14–18
Total	115	18	31	13	6	12	0	208	57	50	23	8	19	**7**	

Remaining open issues were reported to the issue tracker of each project. To limit the number of reports, we created only one issue ticket for similar defects across a given class or multiple classes. Based on feedback from developers, we classify the originally counted issues as false or true reports. Unconfirmed cases are counted as unknown. Finally, we show how many defects are only found by either version of the tool (see Table 4.2).[6]

5 The results show that our enhancements have no strong impact on the false positive rate. In each case, the number of false positives is slightly larger than the number of actual defects found. We are happy that our work resulted in 19 issues being confirmed, most of which are already fixed in the development version of the given projects. Seven confirmed defects can be attributed to our enhancements of Randoop and were not found by the original tool. This confirms that the improvements in code coverage translate to improvements in defect detection as well.

4.1.4.4 Threats to Validity

The benchmark selection itself is always a threat to validity. We try to counter this by choosing 30 diverse packages ranging from very small to fairly large ones.

In our experimental evaluation, the static analysis phase of our enhanced Randoop adds a small constant overhead to the entire workflow. This slightly favors our tool because default Randoop has no static analysis phase. However, the entire static analysis is kept simple to run within a few seconds to sixty seconds, on the benchmarks that we use. Compared to the 10–50 minutes of the dynamic phase, we consider a few seconds start-up overhead to be insignificant.

Furthermore, Randoop is based on random exploration, and our enhancements build on that while also depending on time intervals. These two factors produce slight deviations across multiple executions, and across different platforms. However, the overall coverage results vary only by about 2 % when all enhancements are used together, as their internal variations tend to cancel each other out, so our conclusions are not affected by this.

[6] In the case of *Apache math*, when checking if a test is found by the other version of the tool as well, we consider the entire test set, not just the initial 20 failed tests.

4.1.4.5 Summary

Our enhancements significantly improve code coverage when used together. Not all enhancements are equally effective in all cases, and we expect that more fine-tuning will improve results. Run-time guidance of testing usually also improves the rate at which code is covered, and contributes to finding more defects than what can be found with default Randoop. The false-positive rate is not adversely affected by our enhancements.

4.2 Testing Multiple Versions

4.2.1 Background and Introduction

Software product line engineering (SPLE) manages a set of reusable program assets. It allows to systematically generate families of products to address a particular market segment or to fulfill a specific mission [69]. As with most software development paradigms, SPLE has to ensure the quality of software products effectively and efficiently. To this end, a variety of software testing techniques have been proposed to test software product lines [13, 17, 48, 64, 81, 99].

Random testing is easy to use, scalable and can be fully automated [66]. To test object-oriented programs, random testing randomly constructs object instances as the receiver and input arguments of the method under test (MUT) to exercise different paths in the MUT [34]. Random testing has been found effective at detecting unknown bugs [66]. However, in the context of SPLs, separately performing random testing on each software product of the same SPL causes redundancies: Features shared by different products are tested repeatedly, without increasing test coverage. Furthermore, it is difficult to reuse test results among different product variants.

We propose to use project centralization to test software product lines, because it can eliminate code redundancies by integrating multiple variants. However, existing project centralization techniques [57, 58], work on a class level. On that level of granularity, multiple version variants are only shared if their classes are identical. To test multiple product variants, more fine-grained project centralization that can share common methods is required.

Therefore, we propose *method-level project centralization*, which unifies and shares methods. As methods are defined in their respective classes, the new tech-

nique merges the classes and handles issues related to fields, methods, and inheritance. In general, the technique shares common code whenever possible, while trying to preserve the behavior of each product variant during testing. We implement the random test case generation using Randoop, a state-of-the-art random testing tool [66]. In our framework, Randoop takes the centralized SPL as input and tests multiple product variants in one run. To evaluate our technique, we have conducted cases studies on 33 product variants generated from three non-trivial SPLs. The results are quite promising: Compared to testing each product separately, random testing on the centralized product achieves a higher test coverage. In most cases a high coverage is achieved more quickly as well.

We focuses on testing multiple product variants of SPLs, the concept and techniques presented in this paper do not depend on domain knowledge of SPLs (such as a feature model). They can generalize to other testing scenarios for multiple similar product variants such as historical program versions from software evolution and co-evolution, and similar code branches produced by the *clone-and-own* approach.

4.2.2 Recent Related Works

While much work on SPL testing and automatic test case generation has been done, work on fully automatic test case generation for multiple product variants from SPLs is limited. In this section, we discuss previous works that are most closely related to our work.

4.2.2.1 Software Product Line Testing

When testing SPLs, test cases can be developed separately for each feature. However, it is necessary to run the prepared test cases on each generated product [70]. Unfortunately, running test cases on each product of an SPL is usually not feasible in practice due to the resource limitation. Several approaches try to reduce the combinatorial product test space by product sampling and in other ways, e. g., by reducing the test executions per product [23].

Product sampling selects a representative subset of products from the valid product space of an SPL and only considers these sampled products for testing. Appropriate sampling strategies aim to fulfill given coverage criteria [13, 17, 64, 81], with N-way combinatorial sampling [64] being the most widely used approach.

Other work uses program analysis to reduce the test executions, by running a test case only on a configuration that influences it [49,81]. Kästner et al. [48] explore the execution strategies of a unit test for all products of an SPL without generating each product in a brute force way. They encode the variability of an SPL either in the testing tool (a white-box interpreter) or in a meta-product that represents all products (combined with black-box testing using a model checker) to simulate test execution on all products.

Compared to these techniques, we use code transformation to combine multiple products from an SPL, improving testing coverage while reducing redundancy in test executions. We need only a set of products as input for testing without requiring provided test cases or domain knowledge on an SPL, such as a feature model.

Attempts at sharing the results of test executions have been made before. Xu et al. [99] use a test suite augmentation technique to test multiple products and investigates the influence of the order in which products are tested. The difference of our approach is that no specific product testing order and provided test cases are required, and code transformation on the products (rather than the test cases) is used to share tests among all products.

4.2.2.2 Randomized Test Case Generation

The critical step in random test case generation for object-oriented programs is to prepare the input objects with desirable object states. Most recent random techniques create the required input objects by *method sequence construction* [21, 43, 66, 100].

JCrasher [21] creates input objects by using a parameter graph to find method input and return type dependencies. Randoop [66] use feedback from previous tests to generate future tests. To reuse existing test case to improve code coverage of random testing techniques, several studies are conducted to study the usefulness of reusing existing test cases as domain knowledge for further test case generation of a new product. OCAT [43] adopts object-capture and replay techniques, where object states are captured from sample test executions, and then used as input for further testing. Palus [100] leverages a trained sequence model from the combined static analysis (for method relevance) and dynamic analysis (for method invocation order) to guide test case generation.

However, these testing techniques are designed to test only a single software

product. They do not share the test results to reduce redundancies when testing multiple products.

4.2.3 Our Approach

Project centralization transforms multiple products into a single project, preserving the behavior of each product. Using project centralization, we can generate test cases for all product variants simultaneously. However, our previous centralization [57, 58] only shares a class among multiple products if it has the same implementation throughout. For SPL testing, a more fine-grained centralization is required to increase code sharing.

4.2.3.1 Method-level Project Centralization

A *project* represents the code of a product variant, which has a unique identifier and a set of classes. Each class has a unique name, a set of fields and methods, along with a set of attributes (super classes, interfaces, etc.), as defined in the class file structure [55].

Method-level project centralization transforms a set of projects, $P = \{p_1, \dots, p_n\}$, into a single project p_{centr} such that each method and its implementation from every $p_i \in P$ is preserved in p_{centr}. Methods from different projects are generally separate from each other, since they are defined in their respective declaring classes. To share as many methods as possible, we adapt our class-level project centralization [57, 58] to individual methods.

When merging classes from different projects, techniques of class-level project centralization are first applied, representing all classes with the same name as a separate set. For each such a set of classes $C = \{cl_1, \dots, cl_k\}$, where each $cl_i \in C$ occurs in P and all of them have the same name, the method-level project centralization merges its classes that satisfy the following conditions:

1. The classes are consistent w. r. t. their class attributes (class versions, super classes, interfaces, etc.).

2. All fields with the same name are consistent. A field $cl.f$ of class cl is consistent with another field $cl'.f'$ of class cl', if $cl.f$ and $cl'.f'$ have the same type, annotations, and attributes.

3. All methods with the same name and descriptors are consistent. Method consistency is similar to that of fields, except that the method bodies may be different. In particular, the static initializer, `<clinit>`, of each class in C needs special treatment, as it is only executed at class load time and cannot be executed more than once, even if the initialization of multiple variants of a class is to be simulated. Static initializers can be considered consistent if they are either identical, or if their method body instructions are totally ordered w. r. t. the subset relation.

Before actually merging any classes, we rename the the classes that do not satisfy the above conditions, marking them as distinct classes from different products and also updating references to these classes accordingly.

After ensuring all the classes are consistent for merging by renaming the inconsistent classes, the method-level centralization starts its core steps to merge common code for all those classes $C = \{cl_1, \ldots, cl_k\}$ that share the same name. The first two steps are: (1) create a centralized class with the same name and attributes as a class $cl_i \in C$, and (2) use the union of all fields of the classes in C to synthesize the set of fields of the centralized class. For all methods $M = \{m_1, \ldots, m_l\}$ that occur in C and have the same name and descriptors, we first partition M into $MP = \{mp_1, \ldots, mp_h\}$ such that all methods in the same partition have the same method body. From each partition, we only keep one *representative method* from the set of identical methods.

If MP has exactly one partition, we simply use its representative method in place of all the methods in that partition. If MP has more than one partition, we create a new *centralized method*, which has the same attributes as all the methods in M except for the method body. We then rename the representative methods from each partition to a new unique name to distinguish the different variants. We also keep the project identities for each representative method to remember which projects it comes from and represents. In the method body of the centralized method, we use a switch statement that checks the project identities and forwards a method invocation of the centralized method to the corresponding renamed method.

Project centralization keeps the *transformation map* of each method and class before and after centralization so that we can easily identify each method in the centralized class belongs to which projects. This allows us to preserve the behavior of each original project by forwarding each method call to the corresponding

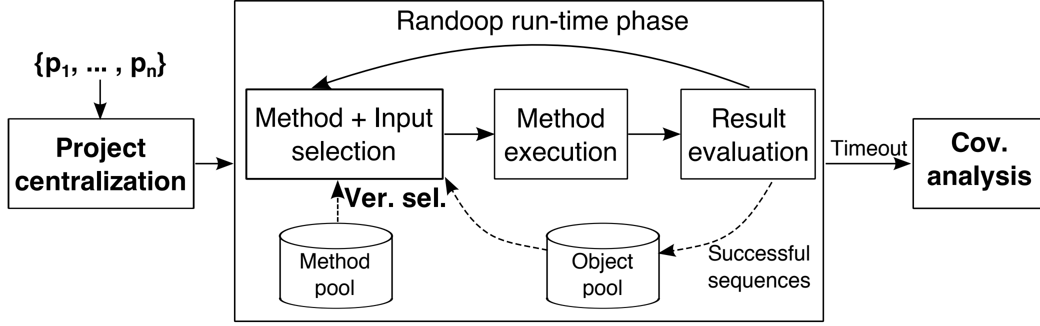


Figure 4.11: Flow of Centralization-based Test Case Generation

renamed method.

4.2.3.2 Integration with Randoop

Randoop [66] is a state-of-the-art random test case generation tool. Given the program to test, Randoop first extracts all publicly visible methods and puts them into a *method pool*, which contains all methods under test (see Fig. 4.11). To test multiple product variants $P = \{p_1, \ldots, p_n\}$ simultaneously, we first perform project centralization on P, and feed both the centralized project and the transformation map into our modified version of Randoop.

Randoop starts by randomly selecting a method $m(T_1 \ldots T_k)$ to test from its method pool. All input objects with type $T_1 \ldots T_k$ must also be prepared to test m. Randoop randomly selects the corresponding inputs from its object pool and concatenates previously known input sequences to derive these objects, to test m. If there is no object with of required type in the object pool, Randoop skips m and selects the next method. Upon successful input construction, m is executed. Execution results of each method call are analyzed against a few predefined contracts. If the generated sequence is new and its execution does not cause any failures, Randoop adds this *successful sequence* to the object pool (see Fig. 4.11). Randoop continues to test more methods until a time limit is hit.

We modify Randoop such that when a method is selected from the method pool, we also randomly set its version, which is represented by the corresponding project identity, before executing the generated test sequences. After the sequence execution, we memorize the selected version for each successful method sequence, so that we can save these generated test sequences as JUnit tests with a specified

version for later use. Instead of re-executing the same method sequence repeatedly, we can change the version of each successful sequence in the object pool to create new sequences, which probably leads to more diverse object states.

After Randoop hits the time limit, we recover the code coverage for each product by analyzing the coverage of each method in the centralized project according to the transformation map of centralization. We also save the generated sequences as JUnit tests.

4.2.4 Case Study

To evaluate our work, we have implemented a tool and applied it to 33 products from three SPLs. Our implementation of method-level project centralization is based on Java bytecode transformation using the ASM library [12], and it is integrated with Randoop and JaCoCo v0.6.4[7] which is used for code coverage analysis. In our case study, we investigate two major research questions:
RQ1: Is project centralization effective in sharing the common code among multiple products?
RQ2: Does testing using project centralization increase code coverage, compared to testing each sampled product independently?

4.2.4.1 Evaluation Subject and Settings

We evaluate our tool on three SPL subjects that were developed and used in previous studies based on FeatureHouse product generation (see Table. 4.3). All the selected SPLs are currently included in the release of FeatureIDE v.2.7.0[8].

Each of these selected subjects is accompanied by both a feature model and source code. For example, GPL has 38 features and 42 constraints in its feature model, which can generate 156 valid products in total by selecting different feature configurations. As pairwise feature coverage is widely used in SPLs as the product sampling approach, we therefore sample valid products with 100% pairwise feature coverage by using SPLATool v0.3 based on the ICPL algorithm [46].[9]

To compare our approach to independent test case generation for each product, we perform project centralization on the sampled products to generate the

[7] `http://www.eclemma.org/jacoco/`
[8] `http://wwwiti.cs.uni-magdeburg.de/iti_db/research/featureide/`
[9] `http://heim.ifi.uio.no/martifag/splcatool/`

SPL desc.	Classes (*.java)	LOC	Features	Constraints	# Products (total)	# Products (pairwise)
Elevator	19	1223	7	3	20	6
GameOfLife	39	1702	23	17	65	8
GPL	57	2957	38	42	156	19

Table 4.3: Case Study Subjects: Size and Complexity Metrics.

centralized project for each SPL. We run our modified Randoop tool chain on the centralized project, while running the original Randoop on each of the sampled product separately.[10] We run each configuration for $1,000$ seconds, after which no noticeable coverage improvement can be observed anymore.

4.2.5 Results

Table 4.4 summarizes the results of our experiments. Column two gives the number of sampled products for each SPL. Columns three and four give the total number of classes (*.class) and methods in all sampled products, respectively. Columns five and six show the corresponding number of *public* classes and *public* methods. Column seven lists the total number of branches. Column eight is the number of classes after performing centralization on all sampled products, while columns nine and ten describe the project size before and after centralization. Finally the average method coverage and branch coverage for all sampled products by both approaches are listed in the next four columns, followed by the total run time of each experiment (the non-centralized cases were run for the full duration in *each* configuration).

In all three cases, centralization shares common code and needs less storage. The centralized project takes 21.6 %, 21.3 %, and 20.3 % of the space required by original sampled products for Elevator, GameOfLife, and GPL, correspondingly. Centralization improves both method and branch coverage, compared to independently testing each project, even though we allot k times the test case generation time to individual testing of k sampled products to have a similar number of test cases in each setting.

[10] Class `GolView` from *GameOfLife* is excluded for testing, because it constantly creates GUI frames that crash the local OS.

Sampled	#Sampled	#All		# Public		#	#classes	Size (KB)		Avg. cov. non-centr.		Avg. cov. centr.		Exec. time (1,000 s)	
pairwise prod.	products	Classes	Methods	Classes	Methods	branches	centr.	non-centr.	centr.	m. [%]	br. [%]	m. [%]	br. [%]	non-centr.	centr.
Elevator	6	90	694	72	552	1658	15	187.0	40.3	88.7	83.9	89.2	88.9	6	1
GameOfLife	8	326	1073	122	833	1636	57	369.3	78.5	59.6	45.4	64.9	53.9	8	1
GPL	19	285	1543	195	1264	1062	21	283.8	57.5	84.7	59.1	86.2	66.8	19	1

Table 4.4: Results of our Experiments. All experiments were run on an Intel Core i7 Mac 2.4 GHz with 8 GB of RAM, running Mac OS X 10.9.3 and Oracle's Java VM (JVM), version 1.7.0_21 with a memory limit of 3 GB for the JVM.

4.2.6 Discussion

To understand the improvement of method coverage, we need to review the test case generation procedure for each method. To test a method, Randoop randomly selects input objects from the object pool. If there is no object with a compatible type, Randoop skips that method. Randoop adopts a fixed method pool. If a method m requires an input object that is not returned by any method in the method pool, m may never be tested. This often happens when testing each sampled product independently. After centralization, however, public methods from multiple sampled products are put into the method pool. This increases the chance to cover more methods, by providing more diverse object types generated by multiple products. Therefore, even if independently testing a product p cannot generate an object typed T to cover a method $m(T\ t)$ in p, $m(T\ t)$ may still be covered by using an object instance typed t from another product p' in the centralization-based testing.

For branch coverage, our approach can use all instances obtained among tests for different products. Therefore, testing a method $m(T\ t)$ can reuse all instances typed T from other sampled products as input. Because of this sharing of test data, more diverse object states from different products are obtained, which improves branch coverage.

However, there exist a few cases where centralization decreases an individual coverage. Centralization increases the method testing space, by introducing more methods and additional product version dimensions. Our current strategy of both method and version selection adopts a uniform distribution, which does not favor common methods by giving them a higher probability. However, the difficulty of covering branches of different sampled products varies. A uniform distribution favors "exploration" (selecting different products) over "exploiting" the same product more thoroughly. A better selection strategy to balance "exploration" with "exploration" is likely to improve the effectiveness of our approach. We leave this as future work.

4.2.7 Threats to Validity

The representativeness of selected subjects is the primary external threat to validity. We carefully select three nontrivial SPLs from different categories that are

widely used in previous studies. We also use their recent implementations based on FeatureHouse. A second external threat to validity is caused by using the default Randoop uniform method and version sampling strategies. Subsequent studies on more advanced strategies and more diverse benchmarks are necessary to decide how our techniques generalize. Another external threat to validity is caused by the randomness of Randoop. We fix and use the same random seed and run each configuration long enough to diminish this threat. The main threat to internal validity is caused by potential bugs of our tool implementation. We decrease this threat by performing unit testing and using the internal verification tool of ASM to check the correctness of code transformation.

4.2.8 Summary

In this paper, we propose method-level project centralization and its integration with random testing to test multiple product variants from SPLs. Our technique shares common code whenever possible, while preserving the behavior of each method for unit testing. The evaluation on three nontrivial SPLs demonstrates the effectiveness of our approach in sharing common code and obtaining higher code coverage, compared to testing each product independently.

Future work includes designing a more advanced strategy to balance shared and unshared code when testing multiple versions. We will also conduct studies on the bug-finding ability of our approach. Furthermore, we are also investigating whether the centralization algorithm can be optimized to share even more code and increase coverage in the given setting.

Chapter 5

Conclusion and Future Direction

5.1 Conclusion

In this thesis, we introduces the version conflict issue in the era of many techniques that facility the development of version variants like revision control tools, SPLs. These version conflicts hinder analysis and verification of multiple product simultaneously by introducing version conflicts. It also brings difficulty to test multiple version variants to improve the overall performance while reducing testing redundancies to ensure the software quality. The overall goal of this thesis is to build an analysis framework and perform verification and testing on multiple version variants to improve software quality efficiently and effectively.

We summarize and formalize the multiple versions and version conflict problems. We proposed the project centralization approach to manage multiple versions while resolving such conflicts. Our technique shares common code whenever possible while preserving the behavior of each version variants. We formalize project centralization and first propose a worklist based algorithm. Then we propose a D-graph representation of projects and transformed project centralization into a graph coloring problem. Based on this representation, we transform project centralization into a graph coloring problem. The corresponding optimal algorithm and heuristic solution are also presented. The experiments on real-world projects demonstrate the effectiveness of our method in sharing common code and resolving

version conflicts, showing the usefulness of our coloring based approach in practice. The further experiment in managing large version variants from software evolution and SPLs further show the effectiveness of our approach.

Based on project centralization, we implement an process centralization tool. We discuss the issues of process centralization and its usefulness in analysis and verifying distributed system with multiple versions. The experiment of our tool on real world distributed application first compares how accuracy of project centralization affects the runtime performance of the centralized program. It also shows the promising direction of our approach in solving the challenging issue of verifying distributed applications, which is a solution that is both sound and complete. And some bugs we find so far is not able to be detected by other approaches like io-cache, which analyze only a single peer.

By realizing the current limitations of automatic software testing tool and limited studies in testing multiple version variants to both share testing results and avoid testing redundancies, we proposed novel program analysis enhanced fully automatic testing techniques and enhanced project centralization based multi-version testing techniques for testing multiple versions to improve software quality in the era of many software versions. Our enhanced testing techniques is fully automatic. It performs both static phase and dynamic phase to make program analysis so that it extracts all necessary domain knowledge from the software under test to further guide testing. The effectiveness and usefulness is proved by fully evaluated on more than 30 real-world benchmarks and SCCH, demonstrating that it can both solves limitations of current techniques in statistical significantly improve the code coverage and improves the defect detection ability. To test multiple software versions, we further refine the accuracy of project centralization by merging class files. We serialize the project centralization results of each version. We further enhance the automatic testing tools and coverage separation mechanism. It uses the project centralization results and code to perform testing, so that the testing results are sheared while avoiding the common code. The case studies of our tool on widely used version variants from software evolution and Software Product Lines demonstrate its usefulness.

5.2 Future Directions

In this thesis, we have performed a consecutive of research studies on software quality insurance techniques in the era of many software versions. We have proposed a project centralization to manage multiple versions. Based on this technique, we have also proposed and implement novel concepts and techniques to verify and test multiple versions to improve software quality.

Multiple versions and version conflict issues recently begin draw researchers attention especially on the study of software evolution and Software Product Lines, where there are many version variants whose quality needed to be ensured before shipping out. However, little studies have been conducted on the fully automatic techniques to manage the multiple version variants to improve overall performance of verification and testing so that both sharing analysis results and avoiding the analysis duplications. We think our approach only gives some initial solutions and tool box for software quality insurance area, although more advanced methods and techniques can be proposed and improved in the future .

We summarize some important future directions based on the studies of this thesis:

- Automatic specification (oracle) mining. Software verification and testing performs program analysis (either static analysis or runtime execution) to check whether the program behaviors is correct and consistent to the specification. However, specification is usually missing for practical applications, which makes the verification and testing very difficult. Therefore, in our verification and testing tool chains, we mostly consider the program failures as the "oracle" to judge whether such an abnormal behavior reveals potential software defects. However, many more program defects that does not fails the software are difficulties to be detected. There exist some recent work on automatic specification minings such as modeling the program behaviors, API protocol minings to show their potential usages to improve verification and testing task. However, it is just at the beginning phase. We think automatic specification mining (includes its some special ones like API protocol) would continue be a hot research topic. The study of mutual enhancement of automatic specification mining and automatic verification and testing techniques would also be very promising, where the specification is learnt during

automatic program execution (through verification or testing) and the learnt specification is further used to guide verification and testing.

- Advanced testing strategy (multi-objective goals). Compared to test a single version, testing multiple versions needs to balanced limited resources on the exploration and exploiting among multiple versions. This can be naturally represented as an multi-objective optimization problem. For example, we would like to achieve the goals like high average coverage, low deviations and so on. Based on our multi-version testing techniques, we can further propose multi-objective whole test suite augmentation towards these predefined multiple goals so that the better decision can be made by selecting the appropriate subset of all generated solutions.

- Automatic verification and testing comprehension. Based on the program failure as specification, we can get many results, including the false positive ones. Current techniques need to manually analyze each of the generated results to identify whether it is the true positive or false positive, which is laborious and error prone. Therefore, the automatic verification and testing result comprehension to automatically classify true positive, false positive and analyze the reason of failure would boosts the application of automatic techniques to more widely and practical applications.

- Novel language and platforms. Many current languages and platforms do not support multiple version, which is one of the reasons make the testing and verification task difficult. If new languages and platforms are designed by taking multiple versions as consideration, it would make some of existing techniques easier to be extended to handle multiple versions. Researchers can only focus on the solution for multiple version itself without spending much effort on resolving the language and platform limitations for multiple versions.

- Analysis strategy for multiple versions. In Software Product Lines research community, researchers classify current techniques into 3 categories: 1. exhaustive approach. 2. sampling approach, 3. family approach. Exhaustive approach generate all versions and analyze each of them. Due to many version in practice, such a solution is usually impractical. Sampling approach tries to sample those version variants that mostly has defects. This approach

is feasible but incomplete. Family based approach represents all product line version variants and analyze them all at once. However, this approach is usually for static analysis and not suitable for accurate run-time behavior analysis for each products. When going beyond Software Product Lines to a more general multiple version cases, we think it is still necessary to design more general techniques for management and analysis of multiple versions. Our work in this thesis gives the basis of behavior preserved multiple versions management. We think more advanced management and analysis techniques are needed to be designed to solve more challenging issues for multiple versions in the near future.

Publication

Refereed Conferences and Transactions:

- Lei Ma, Cyrille Artho,Cheng Zhang, Hiroyuki Sato: Efficient Testing of Software Product Lines via Centralization, Proceedings of 13th ACM International Conference on Generative Programming: Concepts & Experiences (**GPCE'14**), September, Vsters, Sweden, 2014. (submitted)
- Lei Ma, Cyrille Artho, Hiroyuki Sato, Johannes Gmeiner, Rudolf Ramler, Naoto Yokoyama: Enhancing Random Testing through Run-time Guidance, Proceedings of the 29th IEEE/ACM International Conference on Automated Software Engineering (**ASE'14**), September, Vsters, Sweden, 2014. (submitted)
- Lei Ma, Cyrille Artho, Hiroyuki Sato: Project Centralization Based on Graph Coloring, Proceedings of 29th ACM Annual Symposium on Applied Computing (**SAC'14**), pp.1086-1093, Gyeongju, Korea, March 2014.
- Lei Ma, Cyrille Artho, Hiroyuki Sato: Improving Automatic Centralization by Version Separation, Information Processing Society of Japan (IPSJ) Transactions on Programming, Vol.6, No.4, pp.65-77, December, 2013.
- Lei Ma, Cyrille Artho, Hiroyuki Sato: Managing Product Variants by Project Centralization, the 6th International Conference on Computer Science and Information Technology (**ICCSIT'13**), Lecture Notes on Software Engineering, Vol.2 No.2. pp.195 - 200, 21-22, December, Paris, 2013.
- Lei Ma, Cyrille Artho, Hiroyuki Sato: Analyzing Distributed Java Applications by Automatic Centralization, IEEE 37th Annual Computer Software and Applications Conference Workshops (**COMPSACW'13**), pp.691-696, pp.22-26 July, Kyoto, July 2013.

Others:

- Lei Ma, Cyrille Artho, Hiroyuki Sato, Verifying the Distributed System through Automatic Centralization (Oral Presentation), Dependable Component Workshop, Tokyo, September 2012

- Lei Ma, Hiroyuki Sato, An Annotated Type System for Inlining (Oral Presentation), Information Processing Society of Japan(IPSJ PRO 84), Hakotate, June 2011

Bibliography

[1] Saswat Anand, Edmund K. Burke, Tsong Yueh Chen, John Clark, Myra B. Cohen, Wolfgang Grieskamp, Mark Harman, Mary Jean Harrold, and Phil Mcminn. An orchestrated survey of methodologies for automated software test case generation. *J. Syst. Softw.*, 86(8):1978–2001, August 2013.

[2] Sven Apel, Olaf Lessenich, and Christian Lengauer. Structured merge with auto-tuning: Balancing precision and performance. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 120–129, Essen, Germany, 2012.

[3] Sven Apel, J"org Liebig, Benjamin Brandl, Christian Lengauer, and Christian K"astner. Semistructured merge: Rethinking merge in revision control systems. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 190–200, Szeged, Hungary, 2011.

[4] ArgoUML-SPL. http://argouml-spl.tigris.org/.

[5] Cyrille Artho and Pierre-Loic Garoche. Accurate centralization for applying model checking on networked applications. In *Proc. 21st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pages 177–188, Tokyo, Japan, 2006.

[6] Cyrille Artho, Watcharin Leungwattanakit, Masami Hagiya, Yoshinori Tanabe, and Mitsuharu Yamamoto. Cache-based model checking of networked applications: from linear to branching time. In *Proc. 2009 IEEE/ACM Int. Conf. Autom. Softw. Eng.*, pages 447–458, Auckland, New Zealand, 2009.

[7] Luciano Baresi and Matteo Miraz. Testful: Automatic unit-test generation for java classes. In *Proceedings of the 32Nd ACM/IEEE International Conference*

on Software Engineering - Volume 2, ICSE '10, pages 281–284, New York, NY, USA, 2010. ACM.

[8] Garrett Birkhoff. *Lattice theory, the 3rd Edition.* Amer. Math. Soc. (AMS), 1995.

[9] Hans L. Bodlaender and Dieter Kratsch. An exact algorithm for graph coloring with polynomial memory. Technical Report UU-CS-2006-015, Department of Information and Computing Sciences, Utrecht University, 2006.

[10] Uwe M. Borghoff and J. H. Schlichter. *Computer-Supported Cooperative Work: Introduction to Distributed Applications.* Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2000.

[11] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '02, pages 123–133, New York, NY, USA, 2002. ACM.

[12] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[13] Isis Cabral, Myra B. Cohen, and Gregg Rothermel. Improving the testing and testability of software product lines. In *Proc. 14th Int. Conf. on Software Product Lines*, SPLC'10, pages 241–255, South Korea, 2010.

[14] G. Calikli and A. Bener. Empirical analyses of the factors affecting confirmation bias and the effects of confirmation bias on software developer/tester performance. In *Proc. 6th Int. Conf. on Predictive Models in Software Engineering*, PROMISE 2010, pages 10:1–10:11, New York, NY, USA, 2010. ACM.

[15] Everton Cavalcante, André Almeida, Thais Batista, Nélio Cacho, Frederico Lopes, Flavia C. Delicato, Thiago Sena, and Paulo F. Pires. Exploiting software product lines to develop cloud computing applications. In *Proceedings of the 16th International Software Product Line Conference - Volume 2*, SPLC '12, pages 179–187, Salvador, Brazil.

[16] T. Y. Chen, H. Leung, and I. K. Mak. Adaptive random testing. In *Proceedings of the 9th Asian Computing Science Conference on Advances in Computer Science: Dedicated to Jean-Louis Lassez on the Occasion of His 5th Cycle Birthday*, ASIAN'04, pages 320–329, Berlin, Heidelberg, 2004. Springer-Verlag.

[17] Harald Cichos, Sebastian Oster, Malte Lochau, and Andy Schürr. Model-based coverage-driven test suite generation for software product lines. In *Proc. 14th Int. Conf. on Model Driven Engineering Languages and Systems*, MODELS'11, pages 425–439, New Zealand, 2011.

[18] Ilinca Ciupa and Andreas Leitner. Automatic testing based on design by contract. In *Proceedings of Net. ObjectDays*, volume 2005, pages 545–557. Citeseer, 2005.

[19] Ilinca Ciupa, Andreas Leitner, Manuel Oriol, and Bertrand Meyer. Artoo: Adaptive random testing for object-oriented software. In *Proceedings of the 30th International Conference on Software Engineering*, ICSE '08, pages 71–80, New York, NY, USA, 2008. ACM.

[20] Marcus Vinicius Couto, Marco Tulio Valente, and Eduardo Figueiredo. Extracting software product lines: A case study using conditional compilation. In *Proceedings of the 2011 15th European Conference on Software Maintenance and Reengineering*, CSMR '11, pages 191–200, Washington, DC, USA, 2011.

[21] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34(11):1025–1050, 2004.

[22] Laurent Daynès and Grzegorz Czajkowski. Sharing the runtime representation of classes across class loaders. In *Proc. 19th European Conf. on Object-Oriented Programming (ECOOP 2005)*, volume 3586 of *LNCS*, pages 97–120. Springer, 2005.

[23] Ivan do Carmo Machado, John D. McGregor, and Eduardo Santana de Almeida. Strategies for testing products in software product lines. *SIGSOFT Softw. Eng. Notes*, 37(6):1–8, November 2012.

[24] DrJava Project. http://www.drjava.org/.

[25] Tudor Dumitraş and Priya Narasimhan. Why do upgrades fail and what can we do about it?: toward dependable, online upgrades in enterprise system. In *Proc. 10th ACM/IFIP/USENIX Int. Conf. on Middleware*, pages 18:1–18:20, Urbanna, Illinois, USA, 2009.

[26] Tudor Dumitras, Priya Narasimhan, and Eli Tilevich. To upgrade or not to upgrade: impact of online upgrades across multiple administrative domains. In *Proc. ACM Int. Conf. on Object Oriented Programming Systems Languages and Applications (OOPSLA 2010)*, pages 865–876, Reno Tahoe, Nevada, USA, 2010.

[27] D. Faust and C. Verhoef. Software product line migration and deployment. *Software Practice and Experience, John Wiley Sons, Ltd*, 33:933–955, 2003.

[28] Gordon Fraser and Andrea Arcuri. Evosuite: Automatic test suite generation for object-oriented software. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 416–419, Szeged, Hungary, 2011.

[29] Gordon Fraser and Andrea Arcuri. Sound empirical evidence in software testing. In *Proceedings of the 34th International Conference on Software Engineering*, ICSE '12, pages 178–188, Piscataway, NJ, USA, 2012. IEEE Press.

[30] Gordon Fraser and Andrea Arcuri. Whole test suite generation. *IEEE Trans. Softw. Eng.*, 39(2):276–291, February 2013.

[31] P. Godefroid, N. Klarlund, and K. Sen. Dart: directed automated random testing. *SIGPLAN Not.*, 40(6):213–223, June 2005.

[32] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java(TM) language specification, the 3rd Edition*. Addison-Wesley Professional, 2005.

[33] Diwaker Gupta, Sangmin Lee, Michael Vrable, Stefan Savage, Alex C. Snoeren, George Varghese, Geoffrey M. Voelker, and Amin Vahdat. Difference engine: harnessing memory redundancy in virtual machines. *Commun. ACM*, 53(10):85–93, 2010.

[34] Richard Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[35] Health Watcher. http://www.comp.lancs.ac.uk/ greenwop/tao/implementation.htm.

[36] Petr Hnetynka and Petr Tuma. Fighting class name clashes in Java component systems. In *Modular Programming Languages*, volume 2789 of *LNCS*, pages 106–109. Springer, 2003.

[37] Petr Hosek and Cristian Cadar. Safe software updates via multi-version execution. In *Proc. 2013 Int. Conf. on Softw. Eng. (ICSE 2013)*, pages 612–621, San Francisco, CA, USA, 2013.

[38] Wei Huang, Ana Milanova, Werner Dietl, and Michael D. Ernst. Reim & ReImInfer: Checking and inference of reference immutability and method purity. *SIGPLAN Not.*, 47(10):879–896, October 2012.

[39] JaCoCo v0.6.4. http://www.eclemma.org/jacoco/.

[40] Java Runtime Analysis Toolkit. http://jrat.sourceforge.net/.

[41] JavaTM Platform, Standard Edition, V6 API Sepcification. http://docs.oracle.com/javase/6/docs/api/.

[42] H. Jaygarl, C.K. Chang, and Sunghun Kim. Practical extensions of a randomized testing tool. In *Computer Software and Applications Conference, 2009. COMPSAC '09. 33rd Annual IEEE International*, volume 1, pages 148–153, July 2009.

[43] Hojun Jaygarl, Sunghun Kim, Tao Xie, and Carl K. Chang. Ocat: Object capture-based automated testing. In *Proceedings of the 19th International Symposium on Software Testing and Analysis*, ISSTA '10, pages 159–170, Trento, Italy, 2010.

[44] Hojun Jaygarl, Kai-Shin Lu, and Carl K. Chang. Genred: A tool for generating and reducing object-oriented test cases. In *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference*, COMPSAC '10, pages 127–136, Washington, DC, USA, 2010. IEEE Computer Society.

[45] JCarder. http://www.jcarder.org/.

[46] Martin Fagereng Johansen, Oystein Haugen, and Franck Fleurey. An algorithm for generating t-wise covering arrays from large feature models. In *Proc. 16th Int. Software Product Line Conf.*, SPLC '12, pages 46–55, Brazil, 2012.

[47] Jtracert. https://code.google.com/p/jtracert/.

[48] Christian Kästner, Alexander von Rhein, Sebastian Erdweg, Jonas Pusch, Sven Apel, Tillmann Rendel, and Klaus Ostermann. Toward variability-aware testing. In *Proc. 4th Int. Workshop on Feature-Oriented Software Development*, FOSD '12, pages 1–8, Dresden, Germany, 2012.

[49] Chang Hwan Peter Kim, Don S. Batory, and Sarfraz Khurshid. Reducing combinatorics in testing product lines. In *Proc. Tenth Int. Conf. on Aspect-oriented Software Development*, AOSD '11, pages 57–68, Brazil, 2011.

[50] M. Kubale. *Graph Colorings*. Contemporary mathematics v. 352. Amer. Math. Soc. (AMS), 2004.

[51] E. Kuleshov. Using asm framework to implement common bytecode transformation patterns. In *6th International Conference on Aspect-Oriented Software Development*, Vancouver, British Columbia, 2007.

[52] M.M. Lehman and J.F. Ramil. Software evolution in the age of component-based software engineering. *Softw., IEEE Proc.*, 147(6):249–255, 2000.

[53] Zdeněk Letko, Tomáš Vojnar, and Bohuslav Křena. AtomRace: Data Race and Atomicity Violation Detector and Healer. In *Proceedings of the 6th Workshop on Parallel and Distributed Systems: Testing, Snalysis, and Debugging*, pages 7:1–7:10, New York, NY, USA, 2008.

[54] VI Levenshtein. Binary Codes Capable of Correcting Deletions, Insertions and Reversals. *Soviet Physics Doklady*, 10:707, 1966.

[55] Tim Lindholm, Frank Yellin, Gilad Bracha, and Alex Buckley. *The Java virtual machine specification, Java SE 7 Edition*. Addison-Wesley Prof., 2013.

[56] Kasper S. Luckow and Corina S. Păsăreanu. Symbolic pathfinder v7. *SIGSOFT Softw. Eng. Notes*, 39(1):1–5, February 2014.

[57] Lei Ma, C. Artho, and H. Sato. Analyzing distributed Java applications by automatic centralization. In *Computer Softw. and Applications Conf. Workshops*, COMPSACW'13, pages 691–696, Japan, 2013.

[58] Lei Ma, C. Artho, and H. Sato. Project centralization based on graph coloring. In *Proc. ACM 29th Annual Symposium on Applied Computing*, SAC'14, pages 1086–1093, South Korea, 2014.

[59] Darko Marinov and Sarfraz Khurshid. Testera: A novel framework for automated testing of java programs. In *Proceedings of the 16th IEEE International Conference on Automated Software Engineering*, ASE '01, pages 22–, Washington, DC, USA, 2001. IEEE Computer Society.

[60] T.J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2(4):308–320, 1976.

[61] Phil McMinn. Search-based software test data generation: A survey: Research articles. *Softw. Test. Verif. Reliab.*, 14(2):105–156, June 2004.

[62] Thilo Mende, Rainer Koschke, and Felix Beckwermert. An evaluation of code similarity identification for the grow-and-prune model. *Journal of Software Maintenance*, (2):143–169.

[63] T. Mens. A state-of-the-art survey on software merging. *IEEE Trans. Softw. Eng.*, 28(5):449–462, 2002.

[64] Sebastian Oster, Florian Markert, and Philipp Ritter. Automated incremental pairwise testing of software product lines. In *Proc. 14th Int. Conf. on Software Product Lines*, SPLC'10, pages 196–210, South Korea, 2010.

[65] Stefan Paal, Reiner Kammüller, and Bernd Freisleben. Customizable deployment, composition, and hosting of distributed Java applications. In *On the Move to Meaningful Internet Systems*, volume 2519 of *LNCS*, pages 845–865. Springer, 2002.

[66] C. Pacheco and M. Ernst. Randoop: Feedback-directed random testing for Java. In *Companion to the 22nd ACM SIGPLAN Conf. on Object-oriented Programming Systems and Applications Companion*, OOPSLA 2007, pages 815–816, Montreal, Canada, 2007. ACM.

[67] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proceedings of the 19th European Conference on Object-Oriented Programming*, ECOOP'05, pages 504–527, Berlin, Heidelberg, 2005. Springer-Verlag.

[68] Corina S. Pasareanu and Willem Visser. A survey of new trends in symbolic execution for software testing and analysis. *Int. J. Softw. Tools Technol. Transf.*, 11(4):339–353, October 2009.

[69] Clements Paul and Northrop Linda. *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, USA, 2001.

[70] Klaus Pohl, Günter Böckle, and Frank J. van der Linden. *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[71] R. Ramler, D. Winkler, and M. Schmidt. Random test case generation and manual unit testing: Substitute or complement in retrofitting tests for legacy code? In *36th Conf. on Software Engineering and Advanced Applications*, pages 286–293. IEEE Computer Society, 2012.

[72] Brian Robinson, Michael D. Ernst, Jeff H. Perkins, Vinay Augustine, and Nuo Li. Scaling up automated test generation: Automatically generating maintainable regression unit tests for programs. In Perry Alexander, Corina S. Pasareanu, and John G. Hosking, editors, *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, ASE '11, pages 23–32. IEEE, 2011.

[73] Chanchal K. Roy, James R. Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: a qualitative approach. *Sci. Comput. Program.*, 74(7):470–495, 2009.

[74] Julia Rubin, Krzysztof Czarnecki, and Marsha Chechik. Managing cloned variants: A framework and experience. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 101–110, Tokyo, Japan, 2013.

[75] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 218–227, Washington, DC, USA, 2008. IEEE Computer Society.

[76] K. Sen, D. Marinov, and G. Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, September 2005.

[77] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In *Proceedings of the 18th International Conference on Computer Aided Verification*, CAV'06, pages 419–423, Seattle, WA, 2006.

[78] Serp. http://serp.sourceforge.net/.

[79] Nastaran Shafiei and Peter Mehlitz. Modeling class loaders in Java pathfinder version 7. *SIGSOFT Software Engineering Notes*, 37(6):1–5, 2012.

[80] Steven She, Rafael Lotufo, Thorsten Berger, Andrzej Wąsowski, and Krzysztof Czarnecki. Reverse engineering feature models. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 461–470, Waikiki, Honolulu, HI, USA, 2011.

[81] Jiangfan Shi, Myra B. Cohen, and Matthew B. Dwyer. Integration testing of software product lines using compositional symbolic execution. In *Proc. 15th Int. Conf. on Fundamental Approaches to Software Engineering*, FASE'12, pages 270–284, Estonia, 2012.

[82] Scott D. Stoller and Yanhong A. Liu. Transformations for model checking distributed Java programs. In *Proc. 8th Int. SPIN Workshop on Model checking of Software*, pages 192–199, Toronto, Ontario, Canada, 2001.

[83] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, pages 199–215, Berlin, Heidelberg, 2005. Springer-Verlag.

[84] Suriya Subramanian, Michael Hicks, and Kathryn S. McKinley. Dynamic software updates: a VM-centric approach. In *Proc. 2009 ACM SIGPLAN Conf. on Programming Language Design and Implementation (PLDI 2009)*, pages 1–12, Dublin, Ireland, 2009.

[85] Koji Taniguchi, Takashi Ishio, Toshihiro Kamiya, Shinji Kusumoto, and Katsuro Inoue. Extracting sequence diagram from execution trace of Java program. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, pages 148–154, Washington, DC, USA, 2005.

[86] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2):285–309, 1955.

[87] The Java Interactive Profiler. http://jiprof.sourceforge.net/.

[88] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Wolfram Schulte. Mseqgen: Object-oriented unit-test generation via mining source code. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 193–202, New York, NY, USA, 2009. ACM.

[89] Nikolai Tillmann and Jonathan De Halleux. Pex: White box test generation for .net. In *Proceedings of the 2Nd International Conference on Tests and Proofs*, TAP'08, pages 134–153, Prato, Italy, 2008.

[90] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 119–128, New York, NY, USA, 2004. ACM.

[91] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 211–230, New York, NY, USA, 2005. ACM.

[92] Valentin Dallmeier, Christian Lindig, Andreas Zeller. Dynamic purity analysis for Java programs, https://www.st.cs.uni-saarland.de/models/jpure/, 2007.

[93] Joannès Vermorel and Mehryar Mohri. Multi-armed bandit algorithms and empirical evaluation. In *Proceedings of the 16th European Conference on Machine Learning*, ECML'05, pages 437–448, Berlin, Heidelberg, 2005. Springer-Verlag.

[94] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Autom. Softw. Eng.*, 10(2):203–232, 2003.

[95] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. *SIGSOFT Softw. Eng. Notes*, 29(4):97–107, July 2004.

[96] Carl A. Waldspurger. Memory resource management in VMware ESX server. *SIGOPS Oper. Syst. Rev.*, 36(SI):181–194, 2002.

[97] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'05, pages 365–381, Berlin, Heidelberg, 2005. Springer-Verlag.

[98] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for Java programs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '07, pages 75–82, New York, NY, USA, 2007. ACM.

[99] Zhihong Xu, Myra B. Cohen, Wayne Motycka, and Gregg Rothermel. Continuous test suite augmentation in software product lines. In *Proceedings of the 17th International Software Product Line Conference*, SPLC '13, pages 52–61, New York, NY, USA, 2013. ACM.

[100] Sai Zhang, David Saff, Yingyi Bu, and Michael D. Ernst. Combined static and dynamic automated test generation. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ISSTA '11, pages 353–363, Toronto, Ontario, Canada, 2011.

[101] Wujie Zheng, Qirun Zhang, Michael Lyu, and Tao Xie. Random unit-test generation with mut-aware sequence recommendation. In *Proceedings of the*

IEEE/ACM International Conference on Automated Software Engineering, ASE '10, pages 293–296, New York, NY, USA, 2010. ACM.