Improving Software Quality by Synergizing Effective Code Inspection and Regression Testing

Bo Guo
University of Nebraska at Omaha

Improving Software Quality by Synergizing Effective Code Inspection and Regression Testing

By

Bo Guo

A DISSERTATION

Presented to the Faculty of

The Graduate College at the University of Nebraska

In Partial Fulfillment of Requirements

For the Degree of Doctor of Philosophy

Major: Information Technology

Omaha, Nebraska

May 2017

Supervisory Committee :

Dr. Myoungkyu Song

Dr. Mahadevan Subramaniam

Dr. Parvathi Chundi

Dr. Young-Woo Kwon

Abstract of the Dissertation

Improving Software Quality by Synergizing Effective Code Inspection and Regression Testing

Bo Guo, Ph.D in IT

University of Nebraska, 2017

Advisors: Drs. Myoungkyu Song and Mahadevan Subramaniam

Software quality assurance is an essential practice in software development and maintenance. Evolving software systems consistently and safely is challenging. All changes to a system must be comprehensively tested and inspected to gain confidence that the modified system behaves as intended. To detect software defects, developers often conduct quality assurance activities, such as regression testing and code review, after implementing or changing required functionalities. They commonly evaluate a program based on two complementary techniques: dynamic program analysis and static program analysis. Using an automated testing framework, developers typically discover program faults by observing program execution with test cases that encode required program behavior as well as represent defects. Unlike dynamic analysis, developers make sure of the program correctness without executing a program by static analysis. They understand source code through manual inspection or identify potential program faults with an automated tool for statically analyzing a program. By removing the boundaries between static and dynamic analysis, complementary strengths and weaknesses of both techniques can create unified

analyses. For example, dynamic analysis is efficient and precise but it requires selection of test cases without guarantee that the test cases cover all possible program executions, and static analysis is conservative and sound but it produces less precise results due to its approximation of all possible behaviors that may perform at run time.

Many dynamic and static techniques have been proposed, but testing a program involves substantial cost and risks and inspecting code change is tedious and error-prone. Our research addresses two fundamental problems in dynamic and static techniques. (1) To evaluate a program, developers are typically required to implement test cases and reuse them. As they develop more test cases for verifying new implementations, the execution cost of test cases increases accordingly. After every modification, they periodically conduct *regression test* to see whether the program executes without introducing new faults in the presence of program evolution. To reduce the time required to perform regression testing, developers should select an appropriate subset of the test suite with a guarantee of revealing faults as running entire test cases. Such *regression testing selection* techniques are still challenging as these methods also have substantial costs and risks and discard test cases that could detect faults. (2) As a less formal and more lightweight method than running a test suite, developers often conduct code reviews based on tool support; however, understanding context and changes is the key challenge of code reviews. While reviewing code changes—addressing one single issue—might not be difficult, it is extremely difficult to understand complex changes—including multiple issues such as bug fixes, refactorings, and new feature additions. Developers need to understand intermingled changes addressing multiple development issues, finding which region of the code changes deals with a particular issue. Although such changes do not cause trouble in implementation, investigating these changes becomes time-consuming and error-prone since the intertwined changes are loosely related, leading to difficulty in code reviews.

To address the limitations outlined above, our research makes the following contributions. First, we present a model-based approach to efficiently build a regression test suite that facilitates Extended Finite State Machines (EFSMs). Changes to the system are performed at transition level by adding, deleting or replacing transition. Tests are a sequence of input and expected output messages with concrete parameter values over the supported data types. Fully-observable tests are introduced whose descriptions contain all the information about the transitions executed by the tests. An invariant characterizing fully observable tests is formulated such that a test is fully-observable whenever the invariant is a satisfiable formula. Incremental procedures are developed to efficiently evaluate the invariant and to select tests from a test suite that are guaranteed to exercise a given change when the tests run on a modified EFSM. Tests rendered unusable due to a change are also identified. Overlaps among the test descriptions are exploited to extend the approach to simultaneously select and discard multiple tests to alleviate the test selection costs. Although test regression selection problem is NP-hard [78], the experimental results show the cost of our test selection procedure is still acceptable and economical. Second, to support code review and regression testing, we present a technique, called ChgCutter. It helps developers understand and validate composite changes as follows. It interactively decomposes these complex, composite changes into atomic changes, builds related change subsets using program dependence relationships without syntactic violation, and safely selects only related test cases

from the test suite to reduce the time to conduct regression testing. When a code reviewer selects a change region from both *original* and *changed* versions of a program, ChgCutter automatically identifies similar change regions based on the dependence analysis and the tree-based code search technique. By automatically applying a change to the identified regions in an original program version, ChgCutter generates a program version which is a syntactically correct version of program. Given a generated program version, it leverages a testing selection technique to select and run a subset of the test suite affected by a change automatically separated from mixed changes. Based on the iterative change selection process, there can be each different program version that include its separated change. Therefore, ChgCutter helps code reviewers inspect large, complex changes by effectively focusing on decomposed change subsets. In addition to assisting understanding a substantial change, the regression testing selection technique effectively discovers defects by validating each program version that contains a separated change subset. In the evaluation, ChgCutter analyzes 28 composite changes in four open source projects. It identifies related change subsets with 95.7% accuracy, and it selects test cases affected by these changes with 89.0% accuracy. Our results show that ChgCutter should help developers effectively inspect changes and validate modified applications during development.

Keywords: regression testing, extended finite state machines, code review, program differencing, change impact analysis.

Acknowledgments

I would like to express my gratitude to all those who gave me the possibility to finish my Doctorial Degree. First and foremost, I wish to thank my advisors, Professor Myoungkyu Song and Professor Mahadevan Subramaniam for their prolonged, patient and generous supports. It is hard to imagine accomplishing my PhD dissertation without their inspiration and confirmation. Dr. Song and Dr. Subramaniam were always there to listen and to provide critical suggestions. They coached me on how to ask questions and enlightened me to express my ideas precisely. I appreciate them very much for being there at various stages of my research career.

Special thanks go to the rest of my Committee members, Professor Parvathi Chundi and Professor Young-Woo Kwon, for their encouragements, intuitive questions, insightful comments and perceptive suggestions.

I would like to express my sincere appreciation to my Manager, Shalini Rajkumar, at PayPal. Even in cases where I doubted myself, she has always encouraged me and offered me flexible work time, which allowed me to balance work and research time wisely. I would like to appreciate my Directors, Geoffrey Halliwell and Brenda Amber, who provided me an opportunity to work at PayPal. They taught me, by precept and example, how to improve myself and become a resourceful employee. When I faced difficulties within the company, they always provided timely assistance. I also want to thank the PayPal Omaha Scrum team. All of you have been there to support me and make my career smooth and successful.

Finally I would like to express my gratitude to my parents, my wife and my son for their understanding, unconditional support and encouragement to pursue my education. They have given up many things for me to become who I am today. I want to thank them for being there with me every day of my life.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

INTRODUCTION

Software quality assurance techniques are important as high confidence in software system is typically required in software development and maintenance. Although developers spend a significant amount of time and efforts to evaluate a program—*testing*, it is often hard to prevent defects from thriving to system failure and security vulnerabilities [40, 106]. As software bugs are reported, developers spend costly efforts to investigate defective code changes causing bugs—*code review*, while comprehending all associated modifications [6, 92].

Testing and code review compliment each other in development and maintenance to ensure the program correctness, avoiding unpredictable issues in software products. Developers commonly evaluate a program based on complementary techniques: dynamic program and static program analysis. Using an automated testing framework, developers typically discover program faults by observing program execution with test cases that encode required program behavior as well as represent defects. Unlike dynamic analysis, developers ensure the program correctness without executing a program by static analysis. They understand source code through manual inspection or identify potential program faults with an automated tool for statically analyzing a program. To combine static and dynamic analysis, complementary strengths and weaknesses of both techniques can create synergistic analysis. For example, dynamic analysis is efficient and precise but it requires selection of test cases without guarantee that the test cases cover all possible program executions, and static analysis is conservative and sound but it produces less precise results due to its approximation of all possible behaviors but might perform at run time.

Many dynamic and static techniques have been introduced yet testing a program is expensive and inspecting code change is an error-prone process. This research address two fundamental problems in dynamic and static techniques. (1) To evaluate a program, developers are typically required to implement test cases and reuse them. As they create more test cases for new features, the execution cost of test cases increases accordingly. After every modification, they periodically run the existing test cases—*regression test* to ensure the program executes without introducing new faults in the presence of program evolution. To reduce the time required to perform regression testing, developers should select a *safe* subset of the test suite with a guarantee of revealing faults as running entire test cases. Such *regression testing selection* techniques are still challenging as these method also have substantial costs and discard test cases that could detect faults. (2) As a less formal and more lightweight method than running a test suite, developers more often conduct code reviews based on tool support; however, understanding context and change is the key challenge of code reviews. While reviewing code changes—addressing one single issue—might not be difficult, it is extremely difficult to understand complex changes including multiple issues such as bug fixes, refactoring, and new feature additions. For example, a developer, who maintains software versions in the source code management system (SCM), commits her changes grouping multiple bug fixes, feature additions, refactorings, etc. Although such changes do not cause trouble in implementation, investigating these changes becomes time-consuming and error-prone since the intertwined changes are loosely related, leading to difficulty in code reviews. According to studies, the problem above could be mitigated by decomposing tangled changes into related change subsets [9, 46, 92].

To address the limitations outlined above, this research makes the following contributions.

1.1 Model-Based Regression Test Selection

We introduce a model-based regression test selection approach to efficiently building a regression test suite that facilitates extended finite state machines (EFSMs). We consider EFSMs supporting a rich set of commonly used data types including booleans, numbers, arrays, data queues, and record data types. Tests for an EFSM are a sequence of input and expected output messages with concrete parameter values over the data types supported by the EFSM. Changes to the EFSMs are specified at the transition level and add/delete/replace one or more EFSM transitions. Given a change, and a test suite, our approach automatically analyzes each test description in the given test suite to provably predict whether or not the test will exercise the change when it is run on the modified EFSM. It constructs a regression test suite entirely by selecting the tests that will exercise the change. Tests are not actually run on the EFSMs for selection.

We introduce a class of *fully-observable* tests. Informally, a test is fully-observable if all the transitions that will be executed when the test is run on the EFSM can be determined a priori by analyzing the test description. We formulate an invariant for each test such that the invariant is a satisfiable formula if and only if the test is fully-observable. The invariant for a test is automatically built using the transitions (and their post-images) *matching* the test description. Informally, a transition is a match for a test description if it can process some test input in the description. A theorem prover is used in a push-button way to determine if the invariant is satisfiable and identify fully-observable tests.

Essentially, the invariant for a test describes all the plausible EFSM execution paths that the test can potentially take when it is run on the EFSM[1]. In general, the invariant for a test can be large since it encodes several EFSM execution paths including several impossible ones. To enable efficient checking of the satisfiability of the invariant by a theorem prover, a compatibility relation over transitions is introduced. The compatibility relation captures the transitions that can immediately follow another transition in all EFSM execution paths. Compatibility information among transitions is automatically pre-computed using a theorem prover. An acyclic, directed graph represents the compatibility information about the transitions matching a test description. An efficient procedure to determine whether or not a test is fully-observable is developed using the compatibility graph. The procedure traverses the graph level by level to incrementally evaluate the invariant to determine whether or not the test is fully-observable.

We also describe incremental procedures [75, 90] that select fully-observable tests exercising added, deleted, and replaced transitions in the EFSM changes. In fact, to accurately predict if a test will exercise a change, it is enough if the test is fully-observable up to

[1]All the EFSMs and their tests in the dissertation are assumed to be deterministic. So, at most one path can be feasible in the invariant of any test. More details on feasible paths are in section 3.4.

positions in the test description where the transitions appearing in the change match the description. These procedures identify all the positions in the test description matched by the transition appearing in the change and check if the test is fully-observable up to any of these positions. If so, then we have complete information about the transitions executed up to these positions and use this information to accurately determine if the change will be exercised and select the test. Then, the procedures incrementally update the compatibility graph for future changes[2].

Certain tests in a given test suite (including those exercising a change) may become *unusable* because their executions on the modified EFSMs fail. Such failures may happen either because the interface of the modified EFSM is different, some test inputs cannot be processed by the modified EFSM, and/or the output generated by the modified EFSM and that of the test do not match relative to the given test purpose. Tests becoming unusable due to interface changes are easy to identify and can be removed regardless of whether or not the tests exercise changes. However, a test execution that exercises a change and subsequently fails on the modified EFSM may still be useful because the failure highlights an adverse impact of the change. In general, automatically determining whether or not such failing tests are useful is a difficult problem since it requires determining the cause of failure. To address this problem, our basic idea is that *a test is executed to exercises a change, and subsequently a failed test is unusable only if the failure can be removed by using the EFSM transitions*. Unusable tests are discarded; however, if the failure of a test cannot be removed, the test is selected for regression to highlight the potential adverse impacts of the change.

[2]The approach can also be generalized to select tests that are not fully-observable. Some preliminary work in this direction can be found in [89].

We describe a simple procedure to identify and remove unusable tests exercising changes; however, tests in the given test suite often comprise overlapping descriptions. For instance, it is typical for tests to use the same inputs to bring an EFSM to a common state and then exercise other specific behaviors. Such tests as well as others can be selected (and discarded) simultaneously whenever a given change matches these tests at the overlapping portions of their descriptions. To analyze a test suite organized into a test forest with overlapping descriptions, we describe a procedure to simultaneously select and discard groups of tests. Such a procedure alleviates regression test selection costs in many cases.

Our approach has been implemented and applied to EFSM models representing protocols, web services, and other applications with encouraging results. Our experimental results based on a well-known regression cost model [78] show that our approach is economical for regression test selection in all these examples.

1.2 Code Review for Composite Changes

In this research, we also present a technique to support code review and regression testing, called ChgCutter. As it is designed for interactiveness, a developer uses ChgCutter to select a sub region of composite changes. ChgCutter, then, automatically (1) decomposes changes of interest using data and control dependence relationships, (2) summarizes related changes by matching decomposed changes against the rest of a program, and (3) automatically applies identified related changes to the original program version to produce an *intermediate* source program version guaranteed to compile and run with test cases.[3]

[3]We use the terms *intermediate program version* and *intermediate version* interchangeably.

An *atomic change*—code changes that tackle one single issue—might not be too difficult to inspect; however, developers using Version Control Systems (VCS) often commit *composite changes*—code changes that intersperse other kinds of multiple development issues—in a single transaction [47, 92, 93]. Although developers often commit composite changes along with explicit commit logs to VCS, for a reviewer who is interested in particular development issues such as inspecting bug-fixes or analyzing the impact of feature addition or refactoring, it is hard to understand which change regions are related to individual issues.

Tao and Kim [93] empirically studied on the occurrence of composite code changes in four open source projects and found that 17% and up to 29% of the revisions are composite changes addressing more than one issue. Herzig and Zeller [47] manually investigated six open source projects and found that composite changes occur frequently (up to 15%) and appear unrelated as one atomic change. Barnett et al.'s study on the software (Bing and Office) at Microsoft [9] also found that over 40% of changes submitted for code reviews can be potentially decomposed into multiple atomic change sets.

Although developers can investigate changes using software versioning and revision control systems (e.g., SVN and Git), it is not easy to use these tools to search for related changes or revisions that interfere with or depend on one's own changes according to a various notion of relevance. Previous research efforts have focused on *untangling* composite changes. Herzig and Zeller [47] presented an untangling change algorithm and provided evidence that composite changes can affect research in mining software repositories. Barnett et al. [9] introduced a technique for untangling composite changes and identifying independent parts of changes. However, developers are still burdened with the task of understanding and applying partitioned change sets to the original version. Moreover, they very often need to build a syntactically valid version separated from the latest version that combines composite changes to determine the location of a change that has caused a failure during regression testing.[4]

The major goal of this research is to partition composite changes into related change sets and to generate an intermediate program version. Also, we cost-effectively validate software changes applied in the intermediate versions by utilizing a test selection technique.

1.3 Research Agenda

In this research, we address the following research questions in light of the challenges of testing and code review in the process of validating and comprehending an extensively modified program.

- **RQ1**: Can we aid testers build regression test suites all of whose tests are provably guaranteed to exercise a change to a system?

  To maintain an updated software system, all changes to a system must be comprehensively tested for users to gain confidence that the modified system behaves as intended. Generally developers periodically *regression test* by re-running the existing tests to provide confidence that its changes does not impact the existing functionalities. Re-running all tests in the test suite for a complex system may require an unacceptable amount of time and efforts [39]. To reduce the regression time, *Regression test selec-*

[4]In the rest of the dissertation, we will consider "regression test" as unit tests for the process of validating changed programs.

tion techniques are used to select a subset of the existing test suite and run the selected test cases to validate the changed parts of the system. To safely and effectively reduce the size of the test suite, our test selection approach should verify both original and modified parts of a program without dropping test cases that possibly reveals errors.

- **RQ2**: Is our approach is economical?

  To reduce a substantial amount of time to rerun the test suite, an alternative approach—the regression test selection technique is presented but it is not perfect due to the test selection costs. For the cost-effectiveness of regression testing, we should develop an *economical* approach for the regression test selection, making the cost of the test selection procedure less than the cost for executing and checking the extra previous tests required to retest the test suite. To determine whether our test selection technique is cost-effective relative to existing testing techniques described in [68], we exploit a cost measurement model and have achieved acceptable results.

- **RQ3**: Can our approach accurately construct syntactically valid intermediate versions by decomposing a composite change?

  As code change fragments are randomly selected from composite changes by a code reviewer, they are difficult and almost impossible to decompose automatically, even with the most advanced change partitioning techniques [9, 46], which mostly focus on artificially mixed changes or only small sample sets. To make things worse, unlike sampled changes, user-selected changes during code reviews are usually syntactically incomplete. A clear decomposition analysis of a mixed, complex change set should be performed to help code reviewers understand changes easier and detect defects quickly. To make an *intermediate program version* only including separated, dependent changes be executable under a test suite without runtime error or termination, our approach should analyze each change region and its surrounding context based on static program analysis and program transformation techniques.

- **RQ4**: Can we validate program changes by accurately selecting a subset of the regression tests to validate each intermediate version?

  To the best of our knowledge, no tool has ever been proposed to automatically validate the correctness of program changes by combining the two approaches—testing and code review. The developers attention on changes during code reviews has usually different properties from validation of execution in testing. They are required to know the failure reasons, being eager to understand all related context causing higher critical issues, in addition to failure detection. Our hybrid analysis techniques should be applied to a single problem in tandem to complement and support one another, leveraging complementary strength of testing and code review. In other words, a code review tool integrated with a testing tool will build a decomposed intermediate version, which will later be fed to a regression testing tool that should analyze coverage to guide test selection. The result for this question can be evaluated by inspecting accuracy how closely selected test cases affected by changes match with the expected result—the number of correctly selected test cases to test an intermediate version

1.4 Major Research Contribution

In this research, we address and provide solutions for the following research questions in light of the challenges of testing and code review in the process of validating and comprehending an extensively modified program.

1.4.1 Test Selection for Changes (RQ1)

In the research, we focus on the regression test selection problem on Extended Finite State Machines (EFSMs). Changes add/delete/replace EFSM transactions. Tests are a sequence of input and expected output messages. Given a change and a test suite, our approach automatically analyzes each test description in the given test suite to provably predict whether or not the test will exercise the change during the execution on the modified EFSMs. We introduce a class of *fully-observable* tests. A test is fully-observable if its descriptions contain all the information about the transitions executed by the tests. We formulate an invariant for each test such that the invariant is a satisfiable formula if and only if the test is fully-observable. The invariant for a test is automatically built using the transitions (and their post-images) *matching* the test description. Informally, a transition is a match for a test description if it can process some test input in the description. Incremental procedures are developed to efficiently evaluate the invariant and to select tests from a test suite that are guaranteed to exercise a given change when the tests run on a modified EFSM. A theorem prover is used in a push-button way to determine if the invariant is satisfiable and identify fully-observable tests [89].

1.4.2 Time Saving for Test Selection (RQ2)

Several tests in the given test suite have overlapping descriptions. For instance, it is typical for tests to use the same inputs to bring an EFSM to a common state and then exercise other specific behaviors. Such tests as well as others can be selected (and discarded) simultaneously whenever a given change matches these tests at the overlapping portions of their descriptions. To enable analysis of a group of tests, a test suite is organized into a test forest whose each tree represents a group of tests with overlapping descriptions. We describe a procedure to simultaneously select and discard groups of tests. A test suite tree (TST) is built and comprised of a group of tests all starting with a same input. By left-right traversing TST, a group of tests will be selected if their shared node exercises the change.

To reduce test selection costs, our approach reduces the test suit size by automatically analyzing EFSM and test descriptions. We guarantee that test suite size reduction do not harm error detection capability. During the test selection analysis, our approach automatically translates the EFSM expressions into the language of the prover. It then invokes a theorem prover, called *Simplify*, in a push-button manner to check satisfiability of the generated formulas [29]. To estimate the effectiveness of our approach, we apply our technique to practical EFSM models, including representative protocols, web services and other subject applications. The evaluation result shows that our approach is *economical*, demonstrating our approach is able to select test cases and build a smaller size of test suite, compared to the dependency-based technique.

1.4.3 Partitioning Composite Code Changes (RQ3)

To improve developer productivity in code reviews, we present a novel approach to automatically separate related changes from composite changes and interactively compose relevant atomic changes to generate compilable and executable intermediate versions based on the original program. Our approach is demonstrated by implementing a proof-of-concept prototype and integrating it with Integrated Development Environment (IDEs) as plug-in. Our approach allows a code reviewer to select an example program edit in differences between original and changed versions of a program. It automatically decomposes a composite change into atomic changes, groups related changes locations, and generates an compilable and executable intermediate version that only includes relevant changes. As a result, it can help code reviewers (i) easily understand all changes related to selected regions, (ii) detect defects by automatically testing an intermediate version including relevant changes, and (iii) focus developer attention on failure-inducing changes by analyzing and classifying identified related changes if testing failures are reported.

1.4.4 Validating Intermediate Version (RQ4)

To evaluate changes with a regression test suite interactively during code reviews, we present an approach to validating intermediate versions (outputs from our contribution in section 1.4.3) by combining static (i.e., code review) and dynamic (i.e., testing) techniques. Our hybrid method helps developers easily understand and effectively validate changed code fragments, which are only shown locally, leaving developers to guess closely related functionality distributed throughout the system. Our approach applied a testing technique to validate intermediate versions reduces the time needed for testing each intermediate version even after a large amount of program changes, without missing any test that may be affected by related changes. To systematically analyze and validate most critical issues, our approach applies *change impact analysis* for determining the effects of code modifications. Our approach improves programmers productivity by reducing the amount of time and effort in debugging, because it determines a safe approximation of the code changes responsible for test failures as failure-inducing changes during regression testing.

1.5 Outline

The rest of this research is organized as follows. Chapter 2 surveys related work. This can be divided into three main categories: (1) a survey of regression testing, (2) techniques that selecting test cases based on code or model approaches, and (3) techniques that search code changes, decompose an intermingled change set, and build intermediate program versions. Chapter 3 describes test regression selection including (1) a brief overview of EFSM model and the Simplify prover, (2) definitions of EFSM changes, tests and change exercising tests, (3) definition of fully-observable tests and a procedure to identify full-observable tests, (4) the approach to handle multiple tests, and (5) experiment results. Chapter 4 describes our change decomposition, change reconsruction, and regression test selection approaches. The evaluation of the intermediate version generation and test selection techniques with case study applications is also discussed. Chapter 5 concludes with the future work.

CHAPTER 2

BACKGROUND and RELATED WORK

Software testing is the most common way to improve high software quality. Testing activities support quality assurance by executing a program with test cases designed to complete requirements and examining the resulting outputs produced by these test cases. Developers conduct testing for a small piece of code (i.e., unit testing) and for customer validation of a large information system (i.e, acceptance testing). To increase confidence in the correctness and reliability in evolving software, developers frequently perform *regression testing*—the process for validating a changed program to detect whether the changed code region executes as the required specification. A recent study reports that developers consume more than 50% of their efforts and time on testing since they more often implement critical functions in software, which becomes more complex. Many approaches, studying more efficient methods to perform testing, has been introduced for reducing the percentage of development and maintenance costs devoted to testing in practice [13, 53]. Next we will outline the background of testing techniques of evolving software for reduction in cost and improvement in quality.

2.1 Background

Regression Testing. Although developers tested a program at some point, program modification periodically requires them to retest parts of the changed program. The purpose of regression testing is to perform retesting, after changes are made to a previously tested program, to ensure that changes have not adversely affected features, maintaining the same testing coverage as completely retesting the program [42, 73, 102]. In practice, regression testing has been studied with critical issues such as test case revalidation, failure identification, fault identification, modification dependency and test case dependency. In particular, applying a selective approach to regression testing helps developers identify and retest only those parts affected by modifications [78, 79].

Regression Test Selection. Approaches towards regression testing have been broadly classified as being code-based or model-based. Yoo and Harman [105] is a survey on regression test selection and related problems. Rothermal and Harrold [78] and Harrold and Orso [44] are two other surveys emphasizing code-based approaches. Most of these approaches perform control and data flow analysis to determine the difference between original and modified programs and use available test traces to determine if the test should be selected for regression. A framework is proposed in [78] and used to evaluate various code-based approaches in terms of their inclusiveness, precision, efficiency and generality. Inclusiveness measures the number of modification-traversing tests. Model-based approaches use executable models and model-programs instead of actual code to select regression tests. Model-based regression testing has not received much attention in comparison to the code-based approaches. There has been a lot of interest in this area due to the advent of embedded systems such as automobiles [16] and complex component based systems [19]. Regression test selection for EFSMs have been considered earlier in [20, 61, 62]. In [62] Korel et. al, proposed an approach to automatically reduce a given regression test suite by EFSM model dependency analysis based on dataflow techniques. In [20], Chen el. al refined EFSM dataflow analysis of [62] and handle certain types of transition replacements.The work in [61] develops heuristics for test prioritization in regression testing of EFSMs.

Challenges in Testing. As a common way of verification of a program behavior, testing has been widely used in practice presenting a number of advanced techniques that we describe above. It supports developers to achieve the ultimate goal of helping them construct a program with high quality; however, it also has several limitations. For example, although testing demonstrates the presence of software defects, it cannot show the absence of their anomalies without a guarantee of complete test suites. If developers fail to obtain a well-selected test suite, they cannot observe the program execution context required to analyze errors or failures. To seek alternative or complimentary solutions, developers use static analysis techniques to inspect program source code—*code review*. Developers conduct code reviews based on either manual code change inspection or automated approaches that provides static analysis tools for defect detection. We will describe background of static program analysis (e.g., code review) as complementary solution used for proofs of program correctness.

Code Review. Code review is widely used in practice as an important mechanism to improve quality in practice [3,32,34,100]. By using code inspection results, developer achieve initial error reduction. The cost of rework as a fraction of development and maintenance becomes high, when code reviews are not conducted and faults are discovered during testing. As source code in an object-oriented program is especially distributed [67,101], to comprehend a piece of code fragments, the subsequence of method calls across different classes must be examined followed by understanding inheritance hierarchies. In long-lived, large-scale software systems, quality and productivity improvements due to code inspections have been validated in a study [12,83]. Code inspections are an effective means of removing defects since developers are commonly required to figure out how code changes happened. So, the code inspection practice has been widely disseminated in various industries. Compared with finding defects by testing later in the development process, defects can be quickly discovered and fixed during code reviews [33,81].

Challenges in Code Review. It reportedly takes up to 60% of the software engineering effort [24,93] to investigate past and present program modifications made by other developers. It enables developers to find nuanced differences, to remove bugs, and to understand changes in one part of the software having unexpected impact on other parts. To understand changes, developers typically compare two versions of a program and inspect line-level differences, such as diff output (i.e., textual differences between two versions of a program). Developer spend a significant amount of time and effort in understanding the change correctness. Instead of reviewing the entire program, developer often review only the incremental codes changes. But it still time consuming and error-prone since developer need to identify related changes, as they often address multiple development issues to make *composite* code changes, as opposed to *atomic* changes that address one single issue. Developers often combine numerous unrelated changes such as multiple bug fixes, a lot of feature additions, and refactorings. In an manual investigation of four open source projects, Yida Tao and Sunghun Kim found that up to 20% and on average 17% are composite changes mixed with other types of code modifications [93].

2.2 Regression Test Selection

This section discusses several kinds of regression testing techniques focusing on regression test selection. We classify the characteristics into two areas: (1) code-based regression testing approaches and (2) model-based regression testing approaches.

2.2.1 Code-based Regression Test Selection

Code-based approaches work on programs and have been extensively studied earlier. Yoo and Harman [105] conduct a survey on regression test selection and related problems. The above survey discusses many of these approaches in detail. Most of these approaches perform control and data flow analysis to determine the difference between original and modified programs and use available test traces to determine if the test should be selected for regression. While our research focussed on models it is conceivable that some of these techniques can be adapted to work on Extended Finite State Machine (EFSM) models as well. However, these methods do not target precise selection of tests and instead rely on over-approximations such as the lexicographic comparison used in [79] to perform selection. On the other hand, precisely selecting tests is one of the main goals of the approach, which is crucial to build high-confidence test suites.

Rothermel and Rothermel [78] introduce a framework to evaluate various code-based approaches in terms of their inclusiveness, precision, efficiency and generality. Inclusiveness measures the number of modification-traversing tests[1] in T actually included in T'; precision assesses the accuracy of selection by measuring how many tests that do not traverse or reveal modifications are excluded from T'; efficiency measures the cost of computing T' and running the selected tests versus re-testing all of the tests in T; and, generality measures the overall applicability of an approach. Inspired by this prior work and the empirical evidences on evaluation and comparison of existing techniques, we choose appropriate criteria and measures for applications of our approach by determining correspondences between the model and the code changes. For example, we analytically apply our approach with a guide that if a test exercises changes in a model, it also exercises the corresponding changes in the program.

2.2.2 Model-based Regression Test Selection

Model-based approaches use executable models and model-programs instead of actual code to select regression tests. Although these approaches has not received much attention rather than the code-based approaches, There has been a lot of interest in this area due to the advent of embedded systems such as automobiles [16] and complex component based systems [19], where early testing of models can alleviate the validation costs of actual systems. The model-based approaches commonly associate the original and modified program versions P and P' with executable models M and M', respectively. Tests in the original test suite T are generated using the model M or are hand-crafted and can be executed on original executable

[1]Note that these measures are defined for modification-revealing tests in [78]. We use them for modification-traversing tests.

model M or program P. Additional test scripts may need to be used to execute the tests on program P. Model level changes corresponding to the program changes are identified and used to build the test suite T'.

Previous works on model-based regression testing are presented. Briand *et. al* [14, 15] employ UML models to extract changes by comparing two versions of a class, use case or sequence diagrams in a UML design. These changes are then used to classify a test as a obsolete, retestable, and re-usable test case by mapping a test to a complete message sequence in a sequence diagram. Korel *et. al* [60–63, 95] make effective use of EFSM models. These techniques have focussed on regression test selection, test minimization, and test prioritization problems. They analyze each change—elementary modifications (e.g., addition and deletion)—and its control and data dependences on EFSMs to identify parts of the EFSM affected by changes. These techniques then execute tests selected for regression testing to exercise these parts. To reduce the regression test suite size, the interaction between a test and the impacted parts of an EFSM is often used to capture by using existing tests causing several types of same patterns. Based on those approaches, Chen and Ural [20] present an extended approach to deal with replacement changes, in addition to addition and deletion. They encode tests to a sequence of transitions by disregarding the actual test input values, they then analyze these transitions to select related regression test cases. Due to ignored test cases and conservative data-flow techniques, they often produce false positives and negatives so that selected tests do not exercise changes to be evaluated.

Similar to the aforementioned approaches, we present a model-based approach for executable EFSM models to represent stateful programs and protocols. We analyze the transition level changes such as additions, deletions, modifications, and replacements used in previous approaches [20, 61, 62, 95].

In contrast to previous approaches that are often efficient but nonsafe to perform regression testing in the EFSM, there are four major differences in our approach. First, our approach does not miss fully-observable tests that exercise a change in the EFSM. Second, it handles a rich set of data types including booleans, numbers, and aggregates like arrays, data queues, and record data types. Our models for tests allow test cases to have constant values involving all these data types including aggregate data types. Supporting EFSM and test models having such expressive data types allows for better traceability between code, its tests, and their models, one of the important criteria for successful model-based testing. We analyze EFSMs and tests involving such data types because we use a powerful theorem prover like Simplify [29] that supports several decision procedures to reason about these data types and we have further extended the prover to support both message and data queues [41, 88] and integrated with a rewrite engine based on the prover [55]. Third, our approach automatically identifies unusable tests [44, 78] which comprises both obsolete tests due to interface modifications and incomplete tests resulting in output mismatches. Several previous works fail to identify such unusable tests because they ignore the test input values [20, 61, 62]. Lastly, our approach exploits the overlap in test descriptions to simultaneously select or discard tests for building a regression test suite.

2.3 Theorem Provers and Regression Testing

SMT solvers have been used to perform code-based test augmentation techniques in [82, 103]. In [103] the solver *Yices* [1] is used to generate test input values satisfying the differences between original and modified test traces. The use of prover in [82] is similar in that it generates assignments satisfying new paths and satisfy the negation of the original paths. To the best of our knowledge, the approach is perhaps the first attempt at a formal approach using a theorem prover to perform regression test selection problem for the EFSMs. This work builds on our earlier work on formal change impact analysis for EFSMs [41, 88]. There, we developed an approach to identify EFSM transitions impacted by a change by performing selective state exploration starting with the change instead of forward or backward explorations.

Our research uses a well-known, powerful theorem prover called *Simplify* [29] extended with rewrite rules [55]. The prover includes decision procedures for several commonly used data types including booleans, numbers, arrays, data queues, and record data types, which also enables our approach to automatically analyze data rich EFSM and test descriptions. Building high confidence regression test suites by using a prover can also potentially help in other problems such as test suite minimization and test suite augmentation [44].

2.4 Composite Code Change Decomposition

Tao et al. [92] conduct empirical studies and find that developers often create composite changes by combining multiple changes (e.g., bug-fixes and refactorings) in a single commit transaction. They point out that developers mix multiple bug fixes or other kinds of code changes in one check-in. They also observe that developers spend a large amount of time to review others' composite changes. However, separating related change subsets from composite changes is difficult and error-prone. To help developer understand changes during maintenance and development tasks, they claim that better support is needed for decomposing composite changes and determining the risk of these large, complex changes. Our approach was motivated by their findings and observations to help code reviewers understand and test complex code changes efficiently and systematically. Herzig and Zeller manually investigated 7000 change sets, with up to 15% being tangled. At least 16.5% of all source files are incorrectly associated with bug report. They also developed a change decomposition method to separate tangled changes [46]. Their technique uses heuristics such as file distance, change coupling, data dependencies and call-graph. The untangling algorithm uses a set of change operations, added or deleted method calls or method definitions, as input and return a set of change set partitions. Each partition includes changes to resolve one issue, such as bug-fix. However, they use artificially tangled changes that only contains issue fixing change sets. We instead uses real change data set that mines diverse changes from open source projects.

Tao and Kim [93] study four open-source projects and find that up to 29% changes are composite changes. Based on these empirical findings, they present a static analysis tool to partition composite code changes into change subsets as change slices for supporting peer code reviews. They extract a set of changes such as deletion, modification, and addition to identify related changes and produce partitions as output that consist of only changed lines that are related. First, they isolate formatting-only changes by using *diff* tools. Second, they find data and control dependences by using program dependence graphs. Lastly, they identify similar patterns by using a clone detection technique. Barnett et al. present a technique, called ClusterChanges, to separate regions of change from a change set [9]. ClusterChanges uses def-use chain analysis and analyzes differences between different types of change partitions. Our approach differs from these techniques because of two reasons. First, our approach partitions change subsets, while building an intermediate program version that is compilable based on analysis of the dependencies of modification. Second, our approach can automatically test changed contexts with the regression test suite by regenerating an executable, partitioned program.

2.5 Code Search for Code Comprehension and Inspection

As a common practice in software reuse, developers often copy and paste code fragments, while modifying them or deploying copied code without minor adaptation so that software systems entail similar regions of code, called *code clone* [8, 80]. To better analyze and understand such applications, developers often use tools to find similar code fragments syntactically or semantically. To detect duplicate code, clone detection techniques are actively introduced [49,54,66,70,72,97,98]. Johnson pioneered text-based clone detection techniques analyze raw source code by hashing strings per line [50–52]. By using a sliding window technique combining with an incremental hash function, his approach finds sequences of lines containing the equivalent hash value as clone. Token-based clone detection techniques use a lexical analyzer to divide source code into a token sequence [7, 8, 54]. Tokens are analyzed with a hashing functor and a position index for their occurrence in the line. The prefixes of a sequence of symbols are then formed into a suffix tree. In a suffix tree, there can be a common prefix when suffixes share the same set of edges. If two suffixes share a same prefix, which appears repetitively, they are regarded as a clone. Tree-based clone detection techniques parse a program to a parse tree or abstract syntax tree (AST) representation of the source code [35, 96, 104]. Jiang et al. [49] used vectors to approximate ASTs in a Euclidean space. By computing the Euclidean distance metric, their technique clusters related vectors and finds clones. Lee et al. [66] developed a code search technique that focuses on an approximate clone detection, which is scalable yet often produces false positives. Their technique extracts characteristic vectors from source code and generates a multi-dimensional indexing tree structure. It then filters and ranks the index to evaluate clones in order. Lin et al. proposed a tool, called MCIDiff, to identify similar parts from multiple code clone instances [70]. MCIDiff analyzes differential ranges across clone instances by using a longest common subsequence algorithm and finds similar tokens in differential ranges. Chang et al. presented an approach to finding implicit rules from dependence graphs using graph mining [18]. Wang et al. present a code search technique by capturing control and data dependence relationships [98]. They improved their approach by applying semantic topic modeling [97]. Nguyen et al. proposed a graph-based model for representing object usage [72]. All pairs of code elements (e.g., method, field and class) are identified if there are similar object interactions. Candidates are detected using several heuristics to find similarities in implementation code or naming scheme or the same ancestor method/class or the same interface.

Our research well complements the previous approaches because, compared to only code search/comprehension, an unified approach with code change verification leads to much more precise results, directly reducing development efforts to detect similar mistakes or errors. None of these approaches is capable of automatically identifying and separating intermingled source code changes, in order to build an executable program with separated change subsets for runtime verification. As the similar changes or repeated repairs typically produces similar code in practice, our work can be used along with the approach above to find clones as well as validate a group of similar changes automatically to reduce programmers' manual effort.

2.6 Intermediate Version Construction

Chesley et al. present an approach to updating the original program with a change as well as all of its dependent program elements [21]. Based on the affecting changes of a failed affected test case, they incrementally updated the original program to produce an intermediate program version. As opposed to their approach that relies on programmer selections, we automatically identify *related*, affecting change set using an AST-based code search technique, and apply the identified change set to the original version of a program to obtain an intermediate version. Previous approaches [4, 23, 30, 94] presented *recompilation* techniques that generates a program by applying minimal syntactically valid edit. Tichy [94] specified dependent relationships to represent how compilation units are related. Syntactic dependence identification is empirically evaluated, reporting 50% reduction of the recompilation effort [4]. Burke et al. developed an optimization technique during program compilation [23]. Their approach analyzes semantic dependencies between procedures using inter-procedural data flow information such as alias and reference. Dmitriev introduced an approach called *smart dependency checking* to calculate syntactic dependencies for Java class files. If a class changes, its referenced classes are required for recompilation [30]. Similar to the previous approaches, we automatically generate an intermediate version of a program by computing dependencies between change subsets and other related contexts. However, we further optimize an intermediate version, making it possible to be executable without any runtime issue. We then efficiently apply a regression test selection technique to an intermediate version to validate related change sets.

2.7 Interactive Code Reviews for Inspecting Relevant Changes

Zhang et al. [108] present an interactive code review approach, which is called Critics, for inspecting similar, related changes to multiple code locations. Critics summarizes similar changes and detects potential inconsistent or missing changes. We decided to reuse the user interface of Critics, because the user studies with both student and professional developers show that the interactive feature of Critics can improve developer productivity when reviewing system-wide code changes. For example, Critics allows a developer, who wonders if there are other methods that are changed similarly to method `foo`, to select the changed code in the diff patch. Given the selected change, Critics identifies another method `bar` that matches the change of method `foo`. If further investigating may be required for locating other suspicious locations using different identifiers, the user interface of Critics allows her to generalize a *matching* edit script. Based on the script, Critics can identify method `baz` using different identifiers that matches the change of method `foo`. We leverage the similar, related changes identified by Critics in methods `bar` and `baz` to automatically decompose composite code changes and generate an intermediate version with related atomic change sets. The intermediate version generated by our approach only contains the related changes in methods `foo`, `bar`, and `baz` separated from other changes. During the code generation, our approach produces syntactically valid code by analyzing data and control dependencies.

CHAPTER 3

SELECTING TESTS with PROVABLE GUARANTEES

3.1 Motivating Example

Consider a bank web service EFSM depicted in Figure 3.1(a) consisting of 12 transitions t_1-t_{12} depicted in Figure 3.2. Users start by opening an account with a cash amount greater than or equal to a minimum balance amount (*min*), and are given a unique number as account id *id*. The current balance in each account is represented by an array $B[]$ mapping the account id to a non-negative number. Users operate the account by performing deposits and withdrawals. Withdrawals exceeding the current balance are ignored. Those leading to a balance lesser than the *min* value result in a state where withdrawals are ignored and a deposit that brings the balance above the *min* value is only allowed. Accounts accrue a bonus that doubles the current balance provided a specified maximum *max* number of withdrawals (that are not ignored) and deposits succeed in maintaining a balance greater than or equal to the *min* value. The bonus amount is transferred incrementally to the account and no operations are processed during this period. The solid arcs in Figure 3.1(a) are *external* transitions that can be observed by their messages. Others (dashed arcs in Figure 3.1(a)) are *internal* transitions.

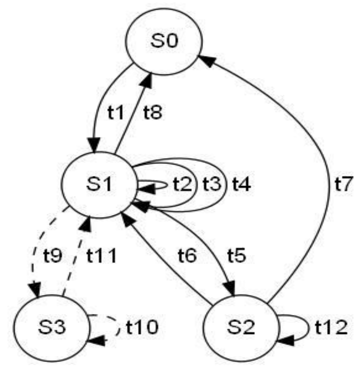
A test suite used to validate the EFSM is given in Figure 3.1(b). Each test starts the EFSM in the state with variables, $id = 0$; $min = 50$; $max = 2$; (Timer) $T1 = max$; $bonus = 0$. It can be verified that each test will run successfully with the EFSM producing the expected outputs.

Now, suppose that we modify this EFSM by deleting the transition t_{12} and adding a transition t'_{12} to allow withdrawals even when the balance falls below *min* as long as a non-negative balance is maintained[1]. Our requirement is to build a regression test suite by selecting tests from the original test suite to validate this change. Usually, tests in the original test suite that execute the newly added transition t'_{12} in the modified EFSM (and/or those that execute t_{12} in the original EFSM) are said to exercise the change and are selected for regression. Test λ_1 is not selected since it can be easily checked that this test executes neither t'_{12} in the modified EFSM nor t_{12} in the original EFSM. The remaining tests λ_2-λ_5 are all selected since we can check that each of them executes the transition t'_{12} in the modified EFSM (and also t_{12} in the original one). However, the test λ_4 is unusable because all its inputs cannot be processed when it is run on the modified EFSM. Similarly, the test λ_5 is unusable because it causes an (unintended) output mismatch when it is run on the modified EFSM. Therefore, the tests λ_2 and λ_3 form the regression test suite for this change.

In all these cases, the descriptions of the tests in the original test suite contain enough information that can be analyzed to accurately predict whether or not a test will exercise the change when it is run. So, these tests can be selected and/or discarded without running them. The approach characterizes tests whose descriptions have enough information and

[1]Note that unspecified inputs in a state cannot be processed and will lead to an implicit dead state.

develops procedures to select and discard such tests based on analysis of their descriptions[2].



(a)

λ1 Open(100)/ack(1), deposit(1, 50)/ack(B[1]), wdraw(1, 160)/ack(B[1]), wdraw(1, 10)/ack(B[1]), close(1)/ack(B[1])

λ2 Open(100)/ack(1), deposit(1, 50)/ack(B[1]), wdraw(1, 110)/ack(B[1]), wdraw(1, 10)/ack(B[1]), close(1)/ack(B[1])

λ3 Open(100)/ack(1), deposit(1, 50)/ack(B[1]), wdraw(1, 110)/ack(B[1]), wdraw(1, 10)/ack(B[1]), deposit(1, 20)/ack(B[1]), close(1)/ack(B[1])

λ4 Open(100)/ack(1), deposit(1, 50)/ack(B[1]), wdraw(1, 110)/ack(B[1]), wdraw(1, 10)/ack(B[1]), wdraw(1, 50)/ack(B[1]), close(1)/ack(B[1])

λ5 Open(100)/ack(1), deposit(1, 50)/ack(B[1]), wdraw(1, 110)/ack(B[1]), wdraw(1, 20)/ack(B[1]), wdraw(1, 20)/ack(B[1]), close(1)/ack(40)

(b)

Figure 3.1: Bank Web Service EFSM and Tests

t1 open(v), (v >= min), s0 → s1, {id += 1; B[id] = v; ack(id)}

t2 deposit(i, v), (i == id ∧ v > 0 ∧ T1 > 0), s1 → s1, { B[id] += v; T1 -= 1; ack(B[id])}

t3 wdraw(i, v), (i == id ∧ v > 0 ∧ (B[i] – v) >= min ∧ T1 > 0), s1 → s1, {B[id] -= v; T1 -= 1; ack(B[id])}

t4 wdraw(i, v), (i == id ∧ v > 0 ∧ (B[i] –v) < 0 ∧ T1 > 0), s1 → s1, {ack(B[id])}

t5 wdraw(i, v), (i == id ∧ v > 0 ∧ (0 <= (B[i] –v) < min) ∧ T1 > 0), s1 → s2, {B[id] -= v; ack(B[id])}

t6 deposit(i, v), (i == id ∧ v > 0 ∧ (B[i] + v) >= min), s2 → s1, {B[id] += v; T1 = max; ack(B[id])}

t7 close(i), (i == id), s2 → s0, {ack(B[id])}

t8 close(i), (i == id), s1 → s0, {ack(B[id])}

t9 (T1 == 0), s1 → s3, {bonus = B[id];}

t10 (bonus > 0), s3 → s3, {B[id] += 1; bonus -= 1;}

t11 (bonus == 0), s3 → s1, {T1 = max;}

t12 wdraw(i, v), (i == id ∧ v > 0), s2 → s2, {ack(B[id])}

t12' wdraw(i, v), (i == id ∧ v > 0 ∧ (B[i] –v) >= 0), s2 → s2, {B[id] -= v; ack(B[id])}

Figure 3.2: Bank Web Service Transitions

[2]Tests executing internal transitions cannot be selected or discarded using their descriptions. For more details, see Section 3.4.

3.2 Preliminaries

This section is mostly derived from earlier works [26, 29, 65, 88], where more details can be found.

Extended Finite State Machines: An extended finite state machine (EFSM) is a finite state machine extended with variables and communicates with the environment by exchanging parameterized messages over (possibly infinite) FIFO queues. An EFSM $E = (I, O, S, V, T)$ [65], is a 5-tuple where I, O, S, V, and T are finite sets of parameterized input and output messages, states, variables, and transitions respectively. Each message in I and O is parameterized and the parameter types are one of – booleans, numbers, arrays, data queues, or record data types. The finite set of variables, $V = X \cup \{IQ, OQ\}$, is the union of the set of data variables X and two message queue variables IQ and OQ denoting the input and output queues from and to the environment respectively. An EFSM transition, t: $im(\overrightarrow{p})$, P_t, $s_t \mapsto q_t$, $om(\overrightarrow{e})$, A_t, where $\overrightarrow{p} = p_1, \cdots, p_n$ are distinct, typed parameter variables of the input message im, the guard P_t is a conjunction of atomic predicates, the action list A_t is an ordered sequence of assignments, and the output message parameters $\overrightarrow{e} = e_1, \cdots, e_w$ is a list of expressions over variables from V and the input parameters $\overrightarrow{p}$. An atomic predicate is formed by applying relational operators (and, or, not, $==$, $\neq$, $<$, $\leq$, $>$, $\geq$) to expressions of the different data types given above. The input (output) messages im (om) are optional in a transition. Transitions having an input and an output message are called **external** transitions; others are **internal** transitions.

Semantics of EFSMs: The semantics of EFSMs are defined operationally using labeled transition systems. An EFSM **global state**, $g = (x_s == s) \wedge pred$, is a formula whose first conjunct sets the data variable $x_s \in V$, denoting the EFSM local state, to the state value $s \in S$ and the second conjunct *pred* is a conjunction of atomic predicates over other variables from V. The predicate *pred* represents all the possible values that the message queue and the data variables can take in the global state g[3]. An **initial global state** is a global state $g = (x_s == s_1) \wedge pred$ whose first conjunct sets x_s to an initial state value $s_1 \in S$ and the second conjunct *pred* is an **initial predicate** that assigns initial values to the variables of V. In the initial global state, in *pred*, the queue variable OQ has the value $initq$ denoting an empty queue; IQ is some input message of I; variables from X have application dependent values.

A **substitution** [5], $\sigma_i = \{x_1 \leftarrow e_1, \cdots, x_n \leftarrow e_n\}$ is a finite mapping from variables x_i's to the expressions e_i's. Substitutions can be applied to messages, transitions, predicates, and global states to obtain their instantiated versions. Application of the substitution σ_i to one of these objects W, $\sigma_i(W)$, replaces every occurrence of the variables x_i's in W by their corresponding expressions e_i's.

Transition t: $im(\overrightarrow{p})$, P_t, $s_t \mapsto q_t$, $om(\overrightarrow{e})$, A_t, is **enabled** in a global state, $g = (x_s == s) \wedge pred$ if $(s == s_t) \wedge (\sigma(im(\overrightarrow{p})) == IQ.\text{head}) \wedge (\sigma(P_t) \wedge pred)$ is a satisfiable formula. The first conjunct in the formula ensures that the input EFSM state of the transition is identical to that in g. The second conjunct ensures that the message at the head of the input queue in g is an instance of the input message of the transition t. The last one ensures that the instantiated guard of the transition t is satisfied in g[4]. The instances of the input message and the guard are obtained by applying a substitution $\sigma = \{p_1 \leftarrow c_1, \cdots, p_n \leftarrow c_n\}$. The substitution σ

[3]Note that parameters, $\overrightarrow{p}$, do not appear in a global state because conditions on these parameters are expressed as conditions over the corresponding queue variables. Further, since parameters in queues need not be bound to global variables the conditions on the parameters cannot be eliminated by flattening an EFSM.

[4]Note that to execute a transition it must be able to enqueue its output messages also. For brevity, we ignore this condition throughout this dissertation. We will assume output queues to be unbounded, henceforth.

is built by point-wise matching of the input message parameters of the transition with the message arguments at the head of the input queue in the global state. The input parameters for which no corresponding argument is found are left unmapped. Substitution σ is the identity mapping if the input queue is empty.

An **execution step**, $g \rightarrow_t g'$, transitions from global state g to g' using t enabled in g. A **run** $r = g_0 t_0 g_1 \cdots t_{n-1} g_0$ is a sequence of consecutive steps starting and ending in an initial global state.

A global state $g = (x_s == s) \wedge pred$ is a **concrete global state** if all the variables in V are fully-instantiated (with constant values) in $pred$. Concrete global states are used later to define test applications and runs.

General and Concrete and Post Images: Informally, the post-images of a transition describe global states produced as a result of executing that transition. More precisely, the **most general post-image** of a transition, t: $im(\overrightarrow{p})$, P_t, $s_t \mapsto q_t$, $om(\overrightarrow{e})$, A_t, $Mgpos(t)$ = $(x_{s0} == s_t) \wedge (x_{s1} == q_t) \wedge (nP_t \wedge IQ_1 ==$ dequeue$(IQ_0) \wedge IQ_0$.head $== im(\overrightarrow{p}) \wedge OQ_1$ $==$ enqueue$(OQ_0, om(\overrightarrow{ne})) \wedge nA_t)$, is a global state representing all the concrete global states that can result after executing the transition t. In $Mgpos(t)$, nP_t is obtained from the guard P_t by renaming variables to refer to their latest instances [25, 88]; x_{s0}, x_{s1} denote the states, $IQ_0(OQ_0)$ and $IQ_1(OQ_1)$ denote the queues before and after executing t respectively; $\overrightarrow{ne}$ denotes the parameter expressions in the output message using the latest instances of variables; nA_t is a set of equalities obtained from the single static assignment [25, 88] representing A_t.

Example : The most general post-image of t_5 in Figure 3.2 is: $Mgpos(t_5) = (x_{s0} == s_1) \wedge$

$(x_{s1} == s_2) \wedge (i == id \wedge v > 0 \wedge 0 <= (B_0[i] - v) \wedge (B_0[i] - v) < min_0 \wedge T1_0 > 0 \wedge IQ_1 ==$ dequeue$(IQ_0) \wedge IQ_0$.head $== wdraw(i, v) \wedge OQ_1 ==$ enqueue$(OQ_0, ack(B_1[id])) \wedge B_1[id] == B_0[id] - v)$. □

Below, we define concrete post-images to deal with the dynamic behavior of tests and relate the static and the dynamic behavior of a transition by relating their most-general and concrete post-images with respect to a concrete a global state. In the next section, we will extend the concrete post-image of a transition to process a test input. Global states enabling a transition called most general pre-images are described later.

Let $g = (x_s == s_t) \wedge pred$ be a concrete global state. The **concrete post-image** of transition t from g, $Cpos(t, g) = (x_{s0} == s_t) \wedge (x_{s1} == q_t) \wedge (pred \wedge nP_t \wedge IQ_1 ==$ dequeue$(IQ_0) \wedge IQ_0$.head $== im(\overrightarrow{p}) \wedge OQ_1 ==$ enqueue$(OQ_0, om(\overrightarrow{ne})) \wedge nA_t)$, is the concrete global state produced by executing transition t in the global state g. If t is not enabled in g then $Cpos(t, g)$ has the value $false$.

Proposition 1 *The concrete post-image of t from a global state g, $Cpos(t, g) = g \wedge Mgpos(t)$.*

Proof: Follows from the definitions of most general post-image and concrete post-image. □

Simplify Prover: In this dissertation, the theorem prover Simplify [29] extended with rewrite rules and queues [88] is used to analyze tests in a push-button manner. The Simplify prover has been extensively used to perform extended static analysis and model checking of software programs [29, 45]. Usually, quantified formulas called verification conditions are generated from the program and input to the prover. The prover automatically determines the validity of an input formula and returns *valid* if the formula evaluates to true under all the assignments to variables in formula and returns *invalid* otherwise. To check if a formula F is satisfiable, its negation $not(F)$ is input to prover. F is unsatisfiable if the prover returns valid; F is satisfiable if the prover returns invalid. Simplify contains decision procedures for numbers, booleans, equality, partial-orders, and the theory of maps. The theory of maps is used to reason about data types such as arrays and record data types in a push-button manner [29]. We automatically translate the data types (and the operations) of the message parameters and data variables in the EFSM into the language of the prover involving the data types supported by the prover. Arrays (fields of a record data type) are translated into maps from array index data types (field ids) to array (field) element types. We also model queues by maps. More details can be found in [88].

3.3 EFSM Changes and Tests

In this section, our model for changes to EFSMs is described, followed by the model for EFSM tests.

3.3.1 EFSM Change Model

Changes to the EFSM are specified at the transition level. An **addition change**, $\delta = \langle +, t_a \rangle$, adds a new external transition t_a to an EFSM. A **deletion change**, $\delta = \langle -, t_d \rangle$, deletes an existing external transition t_d from an EFSM. A **replacement change**, $\delta = \langle -/+, (t_d, t_a) \rangle$, replaces an existing external transition t_d in an EFSM by a new external transition t_a. Certain transition changes may have larger impacts and can modify the EFSM interface itself. For instance, an addition change can introduce new EFSM messages and states. Similarly, a deletion change can result in the removal of existing messages and states. However, in this dissertation, all the EFSM changes are assumed to produce a new EFSM that is both deterministic and consistent. The preservation of these EFSM properties by a given set of changes can be automatically checked using a theorem prover as described in our earlier work in [87].

3.3.2 EFSM Test Descriptions

An EFSM **test**, $\lambda = \langle g_0, [i_1/o_1, \cdots, i_n/o_n]\rangle$, is a pair whose first component is a concrete global state g_0 and the second component is a finite sequence of test elements of the form: test input/expected test output. Each input (and output) is a sequence of assignments to the message queue and/or data variables. Only constants appear in an input. Both constants and data variables can appear in an output.

Example : The EFSM tests for the bank example are depicted in Figure 3.1(b). Test inputs of all these tests assign a single message having constant parameter values to the input message queue and expected outputs assign constant values and variables to the output message queue. □

Note that for brevity, our test inputs and outputs only refer to the message queues and not to the data variables. In fact, we only specify the messages and the queues are implicit. However, our approach is equally applicable to tests having more general inputs and outputs with assignments to data variables. Now, we extend EFSM execution steps and runs to handle test inputs.

Consider test input i: $[m(c_1, \cdots, c_n)]$, adding message m with constant parameters c_1, $\cdots$, c_n to the head of queue IQ. Let g = $(x_s == s_t) \wedge IQ.\text{head} = m(d_1, \cdots, d_n) \wedge pred'$ be a concrete global state.

The concrete global state g extended by a test input i is the concrete global state g_i = $(x_s == s_t) \wedge IQ.\text{head} == m(c_1, \cdots, c_n) \wedge pred'$. The concrete global state g_i is called the **extended concrete global state**. Essentially, in the extended concrete global state, the message parameters are bound to the constant values c_i's specified by the test input and not the original values d_j's.

Transition t **processes** the test input i in the concrete global state g if the transition t is enabled in the extended concrete global state g_i. Test λ is **applied** to EFSM E by starting E in concrete global state g_0. Transition t_0 enabled in g_0 is executed to generate concrete global state, say, g_1, and the process repeated until no more transitions are enabled in the last generated concrete global state, say, g_m. Extend g_m with the first test input i_1 and execute the enabled transition t_m to generate the state g_{m+1}. Transition enabled in g_{m+1} is executed to produce the next state and the process is repeated. The process terminates on either reaching the initial concrete global state after processing all test inputs or if no progress can be made.

Note that the test inputs are consumed by external transitions, and an execution of the EFSM is comprised of executing an external transition (by consuming a test input), running internal transitions as long as possible, and continuing with the next enabled external transition. Note also that a test application may terminate in a non-initial global state either because some test input cannot be processed or because there are no more inputs to be processed.

Test λ is **complete** on an EFSM E if applying λ to E processes all test inputs of λ and ends in an initial global concrete global state of E. The **test run** of a complete test λ on E, $r_\lambda = g_0 t_0 \cdots g_m t_m \cdots g_0$, is a run of E produced by applying λ to E. Note that a test run differs from a regular EFSM run in that the concrete global states in a test run where transitions process the test inputs are extended concrete global states. However, each complete test λ has a unique test run r_λ on E since E is deterministic. Further, a transition t_w of r_λ can process a test input i_w of λ only if all the test inputs $i_1, \cdots, i_{w-1}$ are processed by some of the transitions appearing before t_w in the run r_λ.

Here we assume that all tests in the test suite of the original EFSM are complete tests. The notion of complete tests is used primarily to automatically identify unusable tests as described in Section 6. In addition, it simplifies our proofs. The approach can be easily adapted to handle EFSMs and tests whose executions terminate in some final (or other stable) states. Our notion of complete tests can be realized in practice by incorporating an implicit reset action after each non-complete test execution that brings an EFSM to an initial state.

3.4 Fully-Observable Tests

In this section, fully-observable tests are introduced and an invariant formula characterizing these tests is developed. Then, a compatibility relation over transitions is defined and used to develop a procedure to efficiently identify fully-observable tests using a theorem prover.

Definition 1 *Test λ is* ***fully-observable*** *on E if test run r_λ on E contains only external transitions.*

Example : The tests λ_1-λ_5 in Figure 3.1(b) are fully-observable on the bank EFSM. □

It is clear that every fully-observable test on E is also a complete test on E since the application of the test on E must produce a test run for the test to be fully-observable. However, a complete test on E may not be fully-observable on E since its test run can involve internal transitions of E. Note that if λ is fully-observable on E then the transitions $t_1, \cdots, t_n$ in its test run $r_\lambda = g_0 t_1 g_1 \cdots t_n g_n$ must process the test inputs $i_1, \cdots, i_n$ respectively since no other transitions appear r_λ.

A straightforward way to determine if a test is fully-observable on an EFSM is to instrument all the EFSM transitions, apply the test on the EFSM, and analyze the resulting test run. We can view full-observability in terms of the dynamic behavior of a test. Below, we describe how we can determine whether or not a test is fully-observable by statically analyzing its description and the EFSM transitions. The notions of a transition *matching* a test input and *test extended most general* pre- and post- images are introduced. We then describe how for each test we can automatically generate an invariant formula involving the matching transitions and their extended post-images. We prove that a test is fully-observable if and only if the associated invariant is a satisfiable formula.

3.4.1 Matching Transitions and Sequences

Definition 2 *Transition, t: $im(\overrightarrow{p})$, P_t, $s_t \mapsto q_t$, $om(\overrightarrow{e})$, A_t,* ***matches*** *a test input $i = m(c_1, \cdots, c_n)$ using substitution $\sigma = \{p_1 \leftarrow c_1, \cdots, p_n \leftarrow c_n\}$ if* (1). *$\sigma(im(\overrightarrow{p})) = m(c_1, \cdots, c_n)$ and*

(2). *$\sigma(P_t)$ is a satisfiable formula. The substitution σ is called the **matching substitution**.*

The first condition of Definition 2 states that the test input is an instance of the transition input message. The second condition states that uniformly replacing the parameters by the corresponding constant values given in the test input in the guard P_t produces a satisfiable formula.

The matching substitution σ is built by setting the input message parameters of the transition to the corresponding values in the test input.

Example : In Figure 3.2, t_5 matches the input $wdraw(1, 110)$ of test λ_2 using the substitution $\sigma = \{i \leftarrow 1, v \leftarrow 110\}$ since $\sigma(wdraw(i, v)) = wdraw(1, 100)$ and the instantiated guard of t_5 is: $(1 == id) \wedge (110 > 0) \wedge 0 <= (B_0[1] - 110) \wedge (B_0[1] - 110) < min \wedge (T1_0 > 0)$, a satisfiable formula. □

The static match operation in Definition 2 only checks the input message and the guard but ignores the input state of the transition. Hence the operation is conservative i.e., a transition may match a test input but may not able to process that input when the test is actually applied. However, as shown below, the operation will include all the transitions that can process the test input.

Proposition 2 *If a transition t of an EFSM E processes a test input i in a concrete global state g of E then the transition t matches the test input i.*

Proof: Let t: $im(\overrightarrow{p})$, P_t, $s_t \mapsto q_t$, $om(\overrightarrow{e})$, A_t, be the transition of E. Let test input i = $[m(c_1, \cdots, c_n)]$ and the concrete global state $g = (x_s == s) \wedge IQ.\text{head} = m(d_1, \cdots, d_n) \wedge pred'$. The corresponding extended concrete global state using the test input i is $g_i = (x_s ==$

$s_t) \wedge IQ$.head = $m(c_1, \cdots, c_n) \wedge pred'$. Transition t must be enabled in g_i if t can process test input i in g when the test is actually applied. This means that there is a substitution $\sigma = \{p_1 \leftarrow c_1, \cdots, p_n \leftarrow c_n\}$ such that $\sigma(im(\overrightarrow{p})) = m(c_1, \cdots, c_n)$ and the formula $\sigma(P_t)$ $\wedge$ $pred'$ is satisfiable, implying that $\sigma(P_t)$ is satisfiable. The conditions of Definition 2 are met. Therefore, t matches the test input i. □

Several EFSM transitions can match a test input. Let $T(i_k)$ be the (possibly empty) set of all transitions matching the test input i_k. A **matching sequence**, $\phi(\lambda) = [T(i_1), \cdots, T(i_n)]$, of a test λ is a sequence of sets of transitions constructed by point-wise matching of the inputs of the test λ. A **transition sequence**, $\rho = [t_1, \cdots, t_n] \in \phi(\lambda)$ if each $t_k \in T(i_k)$, k = 1 to n.

Example : Matching sequence $\phi(\lambda_2) = [\{t_1\}, \{t_2,t_6\}, \{t_3,t_4,t_5,t_{12}\}, \{t_3,t_4,t_5,t_{12}\}, \{t_7,t_8\}]$ for the test λ_2 in Figure 3.1(b). A transition sequence of $\phi(\lambda_2)$ is $\rho = [t_1, t_2, t_3, t_3, t_7]$. □

3.4.2 Test Extended Most General Images

Recall from Section 3.2 that the most general post-image of a transition t is a global state representing all the concrete global states that can result after executing the transition t. Below, most general pre-image is introduced first. Then, we extend these images to process test inputs.

The **most general pre-image** of transition, t: $im(\overrightarrow{p}), P_t, s_t \mapsto q_t, om(\overrightarrow{e}), A_t, Mgpre(t)$ = $(x_s == s_t) \wedge nP_t \wedge IQ$.head == $im(\overrightarrow{p})$, is a global state representing all the concrete global states in which the transition t is enabled.

The **test extended most general post-image**, $Emgpos(t_w)$, of transition t_w matching test input i_w using the matching substitution σ_w is $Emgpos(t_w) = Mgpos(\sigma_w(t_w))$[5]. It denotes all global states that can be produced by executing the transition $\sigma_w(t_w)$ i.e., an instance of t_w matching the test input i_w. The test extended most general post-image, $Emgpos(t_w) = false$ (empty set of global states) if t_w does not match i_w.

Note that a global state, $g = (x_s == s) \wedge pred$, belongs to $Mgpre(t)$ (above) if $s == s_t$ and the predicate $pred \wedge P_t \wedge IQ.head == im(\overrightarrow{p})$ is a satisfiable formula. Further, for transition t to be enabled in the global state g, g must belong to $Mgpre(t)$. The **test extended most general pre-image**, $Emgpre(t_w)$, of transition t_w matching test input i_w using the substitution σ_w is $Emgpre(t_w) = Mgpre(\sigma_w(t_w))$. It denotes all global states where the transition $\sigma_w(t_w)$ is enabled; $Emgpre(t_w) = false$ if t_w does not match i_w.

Note that given a test and an EFSM, the test extended post-images and pre-images for the transitions can be automatically obtained by generating the matching substitution for each transition and test input pair, which can then be used to produce the corresponding formulas, as described above.

A Structural Invariant: A structural invariant for a test $\lambda = \langle g_0, [i_1/o_1, \cdots, i_n/o_n]\rangle$, given below, can be formulated solely based on test descriptions and the test extended post-images of the matching transitions. The post-images are automatically generated from the test description as explained above. Below, predicate *Init* checks if its argument is an

[5]Note that we define extended post-images for matching transitions only. In all other cases, the matching substitution $\sigma = \{\}$ and the test extended post-image is unsatisfiable.

initial global state and ρ = $[t_1, \cdots, t_n]$.

$$\psi(\lambda) = \bigvee_{\rho \in \phi(\lambda)} Init(g_0 \wedge \bigwedge_{k=1}^{n} Emgpos(t_k)),$$

Each disjunct of $\psi(\lambda)$ corresponds to a transition sequence ρ from the matching sequence $\phi(\lambda)$ and is made of $n + 1$ conjunctions. First conjunct is the concrete global state g_0 in which the test λ starts and the remaining n conjuncts are the test extended post-images of the n transitions in ρ.

Invariant $\psi(\lambda)$ statically obtained using a test description and the EFSM transitions can be related to the behavior of the test application on the EFSM as described below.

Recall from Section 3.2 that the concrete post-image, $Cpos(t_w, g)$, of a transition t_w from a concrete global state g, is the concrete global state obtained by executing the transition t_w in the concrete global state g. The **test extended concrete post-image**, $Ecpos(t_w, g)$, of a transition t_w matching test input i_w using the matching substitution σ_w is $Ecpos(t_w, g)$ = $Cpos(\sigma_w(t_w), g_w)$, where g_w is the concrete global state obtained by extending g using the input i_w. We generalize extended concrete post-image of a transition to a sequence of transitions ρ = $[t_1, \cdots, t_n]$ point-wise matching a sequence of test inputs $[i_1, \cdots, i_n]$ using a sequence of substitutions $[\sigma_1, \cdots, \sigma_n]$ respectively as follows.

Definition 3 *The test extended concrete post-image $Ecpos(\rho, g_0)$ of a transition sequence $\rho \in \phi(\lambda)$ from the concrete global state g_0 over EFSM E is*

$Ecpos([t_1], g_0) = Cpos(\sigma_1(t_1), g_0)$,

$Ecpos([t_1, \cdots, t_n], g_0) = Ecpos([t_2, \cdots, t_n], Ecpos([t_1], g_0)), n > 1$.

Expressions $\sigma_w(t_w)$'s in the Definition 3 above denote transitions obtained by instantiating the transitions t_w's using the matching substitutions σ_w's.

Recall that the constant values in the substitutions σ_w's are obtained using the corresponding test input i_w. Hence in Definition 3, for a transition t_w matching a test input i_w using the substitution σ_w, $Cpos(\sigma_w(t_w), g) = Cpos(t_w, g_w)$, where g_w is the extended concrete global state obtained from the concrete global state g using the test input i_w. Therefore, the extended post-image of the transition sequence ρ, $Ecpos(\rho,\ g_0)$, represents the concrete post-image obtained from the state g_0 with each transition in the sequence processing the corresponding test input in the concrete global state produced by the prefix of the sequence up to (but not including) that transition. If some transition t_w in the sequence is not enabled in the concrete global state $Ecpos([t_1,\ \cdots,\ t_{w-1}],\ g_0)$ produced by its prefix then $Ecpos([t_1,\ \cdots,\ t_w],\ g_0) = \cdots = Ecpos([t_1,\ \cdots,\ t_n],\ g_0)$, $n \geq w$ are all unsatisfiable.

We say that the transition sequence ρ is a **feasible path** (**run**) over an EFSM E for the test λ, if $Ecpos([t_1,\ \cdots,\ t_n],\ g_0)$ is a satisfiable concrete global state over E. For ρ to be a feasible run, the extended concrete post-image should be a satisfiable concrete initial global state i.e., $Init(Ecpos([t_1,\ \cdots,\ t_n],\ g_0))$ must have the value $true$. Therefore, if a sequence of transitions ρ forms a feasible run from the concrete global state g_0 of a test λ then the application of λ will produce a test run in which each transition in ρ will process the corresponding test input of λ.

The information obtained statically from a test λ and a transition sequence ρ belonging to matching sequence $\phi(\lambda)$ can be related to the application of the test λ as follows.

Lemma 1 *Given test $\lambda = \langle g_0, [i_1/o_1, \cdots, i_n/o_n]\rangle$ and $\rho = [t_1, \cdots, t_n] \in \phi(\lambda)$,*

$$Ecpos([t_1, \cdots, t_n], g_0) = (g_0 \wedge \bigwedge_{k=1}^{n} Emgpos(t_k)).$$

Proof: By complete induction on length of ρ. For the base case, let $n = 1$. Since $\rho = [t_1] \in \phi(\lambda)$ t_1 matches the test input i_1 using the matching substitution σ_1. So, by Definition 3 and Proposition 1 it follows that $Ecpos([t_1], g_0) = Cpos(\sigma_1(t_1), g_0) = g_0 \wedge Mgpos(\sigma_1(t_1))$, which is the same as $g_0 \wedge Emgpos(t_1)$ by the definition of extended most general post-images. For the step case, assume that the claim holds for sequences $\rho = [t_1, \cdots, t_k]$ of length less than or equal to k, i.e, $Ecpos([t_1, \cdots, t_w], g_0) = g_0 \wedge Emgpos(t_1) \wedge \cdots \wedge Emgpos(t_w)$, $w \leq k$. Consider the sequence $\rho = [t_1, \cdots, t_{k+1}]$ of length $k + 1$. By Definition 3, $Ecpos([t_1, \cdots, t_{k+1}], g_0) = Ecpos([t_2, \cdots, t_{k+1}], Ecpos([t_1], g_0))$, which simplifies to $Ecpos([t_1], g_0) \wedge Emgpos(t_2) \wedge \cdots \wedge Emgpos(t_{k+1}) = g_0 \wedge Emgpos(t_1) \wedge Emgpos(t_2) \wedge \cdots \wedge Emgpos(t_{k+1})$, by two applications of the induction hypothesis involving a sequence of length k - 1 and a singleton sequence. $\square$

Therefore, the invariant $\psi(\lambda)$ simply checks that the matching sequence $\phi(\lambda)$ contains at least one feasible run from the concrete global state g_0 in which the test λ is applied. Each disjunct in $\psi(\lambda)$ considers a transition sequence from the matching sequence and incrementally checks its feasibility. The transition sequence is a feasible run if the disjunct is satisfiable. Note that since EFSMs are deterministic, each test has at most one test run and therefore, at most one of the disjuncts is satisfiable.

Theorem 3 *A test λ is fully-observable on an EFSM E if and only if $\psi(\lambda)$ is a satisfiable* *formula.*

Proof: Let $\lambda = \langle g_0, [i_1/o_1, \cdots, i_n/o_n]\rangle$ and $\psi(\lambda)$, given above, be the corresponding invariant. ($\Rightarrow$:) Suppose that the test λ is fully-observable on E but the invariant $\psi(\lambda)$ is unsatisfiable. Since λ is fully-observable, it follows by Definition 1 that its test run on E, say, $r_\lambda = g_0 t_1 g_1 \cdots t_n g_n$, contains only external transitions of E. So, the transitions $t_1, \cdots, t_n$ must process the respective test inputs $i_1, \cdots, i_n$. Then, by Proposition 2, a sequence consisting of these transitions $\rho = [t_1, \cdots, t_n]$ must belong to the matching sequence $\phi(\lambda)$. Now, since ρ is built by projecting out all transitions in the test run r_λ starting at g_0, the sequence ρ forms a feasible run from g_0. So, $Init(Ecpos(\rho, g_0))$ is satisfiable formula. Hence $Init(g_0 \wedge Emgpos(t_1) \wedge \cdots \wedge Emgpos(t_n))$ must be satisfiable, by Lemma 1. And, this is possible only if $\psi(\lambda)$ is satisfiable. A contradiction.

($\Leftarrow$:) Now, suppose that the invariant $\psi(\lambda)$ is satisfiable but λ is not a fully-observable test. Since the invariant is satisfiable, exactly one of its disjuncts corresponding to some transition sequence, say, $\rho = [t_1, \cdots, t_n]$, must evaluate to $true$ i.e., $Init(g_0 \wedge Emgpos(t_1) \wedge \cdots \wedge Emgpos(t_n))$ is satisfiable. Since ρ belongs to the matching sequence $\phi(\lambda)$, $Init(Ecpos(\rho, g_0))$ is a satisfiable formula by Lemma 1. Hence the sequence ρ is a feasible run on E from the concrete global state g_0. So, $r_\lambda = g_0 t_1 Ecpos([t_1], g_0) t_2\ Ecpos([t_1, t_2], g_0) \cdots t_n Ecpos([t_1, \cdots, t_n], g_0)$ is a test run of λ on E. Further, r_λ contains only external transitions since ρ belongs to the matching sequence. Therefore, test λ is fully-observable on E. A contradiction.$\square$

The Theorem 3 establishes that the test description of a fully-observable test contains all the information about the transitions that will appear in its test run. So, for such a test, the transition sequences obtained by the syntactic matching operation can be analyzed using a theorem prover to a priori determine the sequence of transitions that will be executed when the test is applied. This will then provide provable guarantees about a test execution without actually executing the test. In the rest of this section, an efficient procedure to identify fully-observable test is described

3.4.3 Identifying Fully-Observable Tests

One way to determine if a test λ is fully-observable is to obtain the matching sequence $\phi(\lambda)$ from the description of λ, formulate the invariant $\psi(\lambda)$ using the transition sequences in $\phi(\lambda)$, and then input $\neg(\psi(\lambda))$ to the theorem prover to check its validity. If the input is valid then the invariant $\psi(\lambda)$ is unsatisfiable and by Theorem 3, λ is not fully-observable; otherwise λ is a fully-observable test.

However, such a naive approach may not be possible since the invariant $\psi(\lambda)$ may be quite large and can be costly to check using a theorem prover. Often, invariants tend to get large due to the excessive branching among the transitions in the matching sets leading to an explosion of paths i.e., there are too many disjuncts in the formula. Such branching occurs because we compute the matching sets for each test input individually without considering the interactions of these transitions with those in the preceding matching sets. We can reduce branching by analyzing the interactions among the transitions. For instance, a certain matching transition may be safely discarded because it cannot follow any of the preceding matching transitions. Below, a compatibility relation among transitions is introduced to capture interactions and used to efficiently check the invariant.

3.4.3.1 Compatibility of Transitions

Given transitions t_i and t_j with input messages $m_i(\overrightarrow{p_i})$ and $m_j(\overrightarrow{p_j})$ respectively, let predicate ξ: (IQ_0 == enqueue($m_j(\overrightarrow{p_j})$, enqueue($m_i(\overrightarrow{p_i})$, $initq$)) be the input message context with the input queue instance IQ_0 having the input message of t_i followed by that of t_j. Compatibility relation over transitions determines if a transition can immediately follow another in the EFSM runs under a given input message context.

Transition t_j is **incompatible** with t_i under a given input message context if t_j cannot immediately follow t_i in any EFSM run. Transition t_j is incompatible with t_i if $\xi \wedge Mgpos(t_i) \wedge Mgpre(t_j)$ is an unsatisfiable formula i.e., the most-general post-image and the most-general pre-image represent disjoint sets of global states under the given context.

Example : In Figure 2, transition t_{12} is incompatible with transition t_6 under the context ξ: (IQ_0 == enqueue($wdraw(i_2, v_2)$, enqueue($deposit(i_1, v_1)$, $initq$))). The most-general post-image, $Mgpos(t_6)$: $(x_{s0} == s_2 \wedge x_{s1} == s_1) \wedge (i_1 == id \wedge v_1 > 0 \wedge (B_0[i_1] + v_1) >= min_0 \wedge T1_0 == max \wedge IQ_1 ==$ dequeue(IQ_0) $\wedge$ IQ_0.head == $deposit(i_1, v_1) \wedge OQ_1$ == enqueue(OQ_0, $ack(B_1[id])$) $\wedge B_1[id] == B_0[id] + v_1$) and the most-general pre-image, $Mgpre(t_{12})$: $(x_{s1} == s_2) \wedge (i_2 == id \wedge v_2 > 0 \wedge IQ_1$.hd = $wdraw(i_2, v_2)$). It can be verified that $\xi \wedge Mgpos(t_6) \wedge Mgpre(t_{12})$ is unsatisfiable.$\square$

Note that in general, to determine compatibility, the input parameters in transitions t_i and t_j are first renamed so that they are disjoint. Hence incompatibility of t_i with t_j is independent of the constraints involving the input parameters.

Transition t_j is **strongly compatible** with t_i under a given input message context if t_j

can immediately follow t_i in every EFSM run under that context. Transition t_j is strongly compatible with t_i if $(\xi \wedge Mgpos(t_i)) \implies PreElim(t_j)$ is a valid formula. The consequent $PreElim(t_j)$ is obtained from $Mgpre(t_j)$ by eliminating conjuncts that are made of only the input parameters of t_j. This ensures that t_j immediately follows t_i regardless of the values of the input parameters. Note that the most general pre-images and post-images must be satisfiable formulas for a consistent EFSM since every transition in such an EFSM has a satisfiable guard and must terminate. Hence the above formula cannot be vacuously true.

Example : In Figure 2, transition t_8 is strongly compatible with t_6. Context ξ: (IQ_0 == enqueue($close(i_2)$, enqueue($wdraw(i_1, v_1)$, $initq$))); $Mgpre(t_8) = PreElim(t_8)$: $x_{s1} == s_1 \wedge IQ_1$.head = $close(i_2)$; and, $Mgpos(t_6)$ is as given above. It can be verified that $(\xi \wedge Mgpos(t_6)) \implies PreElim(t_8)$ is a valid formula. Also, transition t_6 is strongly compatible with transition t_2 since $(\xi \wedge Mgpos(t_6)) \implies PreElim(t_2)$ is a valid formula. However, it is evident $(\xi \wedge Mgpos(t_6)) \implies Mgpre(t_2)$ is not a valid formula since the conjunct in $Mgpre(t_2)$ obtained from the condition $v > 0$ of the transition t_2 involving the free input parameter v, is not valid. $\square$

It should be emphasized that more than one transition can be strongly compatible with a given transition under a given context provided these transitions have mutually exclusive guards. For instance, consider a deterministic EFSM having three transitions, t_1: m_1, $true$, $s_0 \mapsto s_1$, ack; t_2: $m_2(v)$, $v > 0$, $s_1 \mapsto s_1$, ack; and t_3: $m_3(v)$, $v < 0$, $s_1 \mapsto s_1$, ack. It can be verified that both t_2 and t_3 are strongly compatible with t_1 under the corresponding input message contexts. However, only one of t_2 or t_3 can immediately follow t_1 in any concrete global state under that context. Note however, that transitions whose guards are not mutually exclusive cannot be strongly compatible with the same transition since the EFSMs are deterministic.

Transition t_j is **compatible** with t_i under a given input message context if t_j can immediately follow t_i in some EFSM runs but not in others in that context. Transition t_j is found to be compatible with t_i if t_j is neither incompatible nor strongly compatible with t_i.

Example : In Figure 2, transition t_3 is compatible with t_1 under the corresponding context, since $\xi \wedge Mgpos(t_1) \wedge Mgpre(t_3)$ is not unsatisfiable and $(\xi \wedge Mgpos(t_1)) \implies PreElim(t_3)$ is not valid. $\square$

Several transitions can be compatible with a given transition. Further, it is also possible to have transitions t_j and t_k such that t_j is strongly compatible and t_k is compatible with a given transition t_i.

Note that the compatibility relation uses the most general pre- and post-images of the transitions only and hence can be pre-computed using the EFSM, independent of the tests. We assume that the compatibility information so computed is available with each EFSM transition using three sets – the incompatible, the strongly compatible, and the compatible sets of transitions.

3.4.3.2 Checking Full-Observability

Given a test $\lambda = \langle g_0, [i_1/o_1, \cdots, i_n/o_n]\rangle$, where the concrete global state $g_0 = (x_s == s) \wedge pred$, we use the pre-computed compatibility information among the EFSM transitions and build a directed, acyclic, transition compatibility graph $TCG(\lambda)$. The graph $TCG(\lambda)$ contains the compatibility information about the transitions belonging to the matching sequence $\phi(\lambda)$.

Nodes of this graph denote occurrences of transitions in the matching sequence and the labeled edges denote the compatibility relations.

The graph has a special *start* node and one node for each occurrence of each transition in the matching sequence $\phi(\lambda)$. Node t_{wj} denotes the occurrence of transition t_w in the j^{th} matching set $T(i_j)$ in $\phi(\lambda)$. The graph $TCG(\lambda)$ is a levelized graph with level 0 having the *start* node and the level k, $k \geq 1$, consisting of nodes representing all the transitions in the matching set $T(i_k)$.

Directed, labeled, edges in the graph connect a node in a level to zero or more nodes in the next level. Edges can have a label s (strong compatible) or a label c (compatible). An edge $(t_{vk}, t_{w(k+1)}, s)$ (resp. $(t_{vk}, t_{w(k+1)}, c)$) from node t_{vk} at level k to node $t_{w(k+1)}$ at level $k+1$ with label s (resp. c) belongs to $TCG(\lambda)$ if transition t_w is strongly compatible (resp. compatible) with the transition t_v. No edge exists between incompatible nodes at the successive levels and also between nodes in the same level. An edge from *start* to a node t_{w1} with label s is added if $g_0 \implies PreElim(t_w)$ is a valid formula.

As discussed in the previous section, two transitions can be strongly compatible with a single node only if they have mutually exclusive guards. Since the guards of all transitions matching a particular test input are satisfied under the same matching substitution, they cannot be mutually exclusive. Hence no two transitions in any matching set can be strongly compatible with the same transition. Consequently, in the graph $TCG(\lambda)$, there can be at most one outgoing edge from a node t_{vk} at level k with label s to any of the nodes in the level $k+1$.

```
Inputs: Test $\lambda = \langle g_0,[i_1/o_1, \cdots, i_n/o_n]\rangle$,
        $g_0 = (x_s == s_0) \wedge pred$;
        Graph TCG($\lambda$) with $n$ levels
Output: $Success$ if $\psi(\lambda)$ is satisfiable;
      $Fail$ otherwise
Method:
  $L(start) = true$;
  $Emgpos(start) = g_0$;
  $clevel = 0$; Mark $start$
  while ($clevel < n$) do
    for the marked node, $t_w$, in $clevel$ do:
    Clean-up:
      if $(t_w, t_{w+1}, s) \in TCG$ then
        $TCG = TCG - (t_w, t_v, c)$,
                    $t_v \in TCG$
        Delete dangling nodes and edges
        until none remain
    Extend to next level:
      foreach $t_{w+1} \in immedsucc(t_w)$ do
        $F = (L(t_w) \wedge Emgpos(t_w))$
               $\Longrightarrow Emgpre(t_{w+1})$
        if $valid(F)$
        $L(t_{w+1}) = L(t_w) \wedge Emgpos(t_w))$;
        Mark $t_{w+1}$;
        $clevel = clevel + 1$
      If no $immedsucc(t_w)$ marked, $Fail$
  Let $t_n$ be the last marked node
  if $Init(L(t_n) \wedge Emgpos(t_n))$
  return $Success$
  else return $Fail$
```

Figure 3.3: Full-Observability Procedure

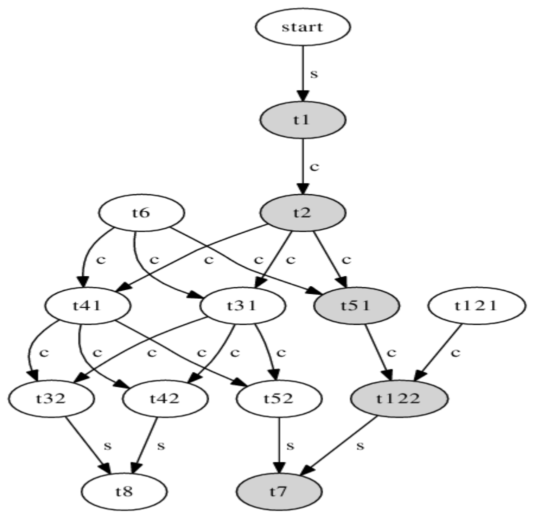


Figure 3.4: TCG for Test λ_2

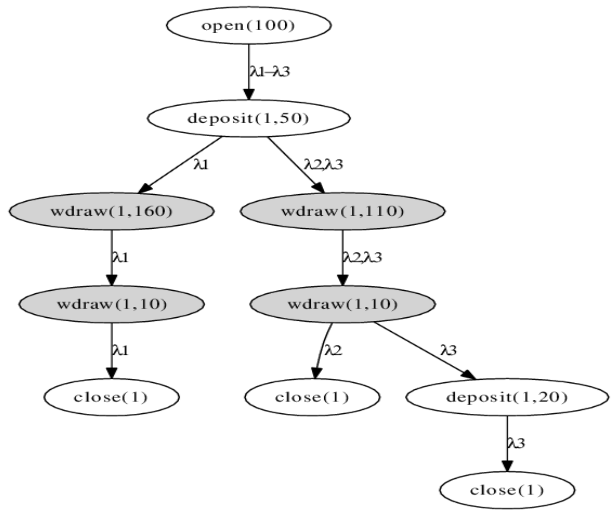


Figure 3.5: Test Suite Tree for Bank Example

Example : The Figure 3.4 depicts the compatibility graph $TCG(\lambda_2)$ for the test λ_2. □

A procedure to determine if a test λ is fully-observable using $TCG(\lambda)$ is described in Figure 3.3. The procedure takes a test λ and its compatibility graph $TCG(\lambda)$ as its inputs and outputs *Success* if λ is fully-observable and *Fail* otherwise. Henceforth, we slightly abuse the notation and refer to the compatibility graph nodes by the corresponding transition.

The graph $TCG(\lambda)$ is traversed level by level starting at the level 0 to see if any path in the graph forms a feasible run from the concrete global state g_0. At each level, a node is **marked** by the procedure if the sequence of transitions in the path from the *start* node to the marked node in the graph form a feasible path from the concrete state g_0 of the test λ. A **label**, $L(t_k)$, is associated with the marked node t_k in each level k. Let $\rho = [t_1, \cdots, t_w]$ be the transition sequence that forms a feasible path from g_0 when the procedure reaches the current level (*clevel* in Figure 3.3 w. The label associated with a node t_k belonging to the sequence ρ is $L(t_k) = g_0 \wedge Emgpos(t_1) \wedge \cdots \wedge Emgpos(t_{k-1})$.

To extend the feasible path to the $w + 1^{th}$ level, first, the candidate immediate successor nodes of the marked node at the current level are identified. If the marked node corresponding to t_w has a strongly compatible immediate successor then this is the only candidate immediate successor as explained above. In this case, the other immediate successors of the node corresponding to t_w linked by c labeled edges, if any, are deleted from the graph. Deleting these edges from the graph may make certain nodes dangling i.e., nodes that do not have any successor and/or a predecessor. Such nodes cannot participate in the feasible run, if any, and hence are deleted from the graph along with the resulting dangling edges.

Such deletion of dangling nodes and edges is repeated until no more such nodes and edges remain. If the marked node corresponding to t_w has no strongly compatible immediate successor then all the immediate successors linked by edges with the label value c are candidate immediate successors.

Then, we propagate the label $L(t_w)$ to each of the candidate immediate successors t_{w+1} and compute a formula $F = (L(t_w) \wedge Emgpos(t_w)) \implies Emgpre(t_{w+1})$. The formula F states that processing the test input w using transition t_w in the concrete global state $g_0 \wedge Emgpos(t_1) \wedge \cdots \wedge Emgpos(t_{w-1})$ will result in a concrete global state in which the immediate successor transition t_{w+1} processing the test input i_{w+1} is enabled. If F is valid then that immediate successor node is marked and the label $L(t_{w+1})$ is set. The current level is incremented and the procedure is continued. If the propagated formula F is not valid for any of the candidate successors then this implies that no transition in the matching set $T(i_{w+1})$ can process the test input i_{w+1} in the concrete global state obtained after processing the first w test inputs. And, the procedure fails. If the path can be extended up to the last level (level n) of the graph and the $L(t_n) \wedge Emgpos(t_n)$ is an initial concrete global state then the procedure returns *Success*. Otherwise, the procedure returns *Fail*.

Example : To determine if the test λ_2 is fully-observable, the graph $TCG(\lambda_2)$ in Figure 3.4 is constructed using the pre-computed compatibility information and analyzed level by level starting at the level 0. The candidate immediate successor for the *start* node is the node t_1. The formula generated at node t_1 is: $F = (x_s == s_0 \wedge (min == 50) \wedge IQ_0 ==$ enqueue$(open(100), initq)) \implies (x_s == s_0 \wedge IQ_0.$head$ == open(100) \wedge 100 > 0)$, is valid. Hence node t_1 in level 1 is marked and hence the label $L(t_1)$ is assigned to be this formula.

The candidate successor at level 2 is node t_2. The formula generated for this node is also valid resulting in the marking of t_2 and its label is set to the generated formula.

At the third level, t_2 has three successors t_{31}, t_{41} and t_{51}. The formula generated at t_{31} is: $F = (L(t_2) \wedge Emgpos(t_2)) \implies Emgpre(t_{31})$ where substitution σ_2 uniformly replaces i by 1 and v by 50 in t_2 and σ_3 uniformly replaces i by 1 and v by 110 in t_3. Since a conjunct in the consequent of F: $((B_1[1]$ - 110) $>=$ 50) cannot be established from the conjuncts: $(B_1[1]$ == $B_0[1] + 50)$ and $B_0[1]$ == 100 in the antecedent of F, the formula F is not valid and the node t_{31} is skipped. Similarly, node t_{41} is also skipped. The relevant conjunct in the formula generated for t_{51} : $(B_1[1]$ - 110 $<$ 50), follows from those in the antecedent, resulting in a valid formula. Hence t_{51} is labeled with this formula as its label $L(t_{51})$. The next two levels have a single successor, nodes t_{122} and t_7 respectively. The formula generated at t_{122} using $L(t_{51})$ is a valid formula and is assigned to $L(t_{122})$. The formula generated at t_7 is also valid and is assigned to $L(t_7)$, resulting in the executable path that is highlighted in Figure 3. Therefore, the test λ_2 is declared fully-observable, which can be easily verified. $\square$

3.5 Selecting Fully-Observable Tests

The regression test selection problem for the EFSMs is analogous to that for programs [44]. It takes as inputs – a deterministic, consistent EFSM E_1 with a test suite T, and a change δ that produces a modified, deterministic and consistent EFSM E_2. It outputs a test suite $T' \subseteq T$, consisting of subset of tests of T guaranteed to exercise the change δ on E_2. All the tests in the original test suite T are assumed to be complete on E_1.

Test $\lambda = \langle g_0, [i_1/o_1, \cdots, i_n/o_n]\rangle$ exercises an addition change $\delta = \langle +, t_a\rangle$ on E_2 if there exists a feasible path $\rho = [t_1, \cdots, t_a]$ from g_0 on E_2. Test λ exercises a deletion change $\delta = \langle -, t_d\rangle$ on E_2 if there exists a feasible path $\rho = [t_1, \cdots, t_d]$ from g_0 on E_1. Test λ exercises a replacement change, $\delta = \langle -/+, (t_d, t_a)\rangle$ on E_2 if it either exercises the addition change $\langle +, t_a\rangle$ on E_2 or it exercises the deletion change $\langle -, t_d\rangle$ on E_2.

Our criteria for tests exercising changes are inspired by the notion of modification-traversing tests described in [78]. Note that a modification-traversing test may or may not produce identical observable behaviors on both E_1 and E_2. Such tests are anyhow included for regression. But if a test is not included for regression then its observable behavior on these two EFSMs will be identical.

For test selection, we slightly generalize fully-observable tests. Let λ be any test for an EFSM E.

Definition 4 *Let $\rho = [t_1, \cdots, t_k]$ be a transition sequence obtained from a path start $\rightarrow t_1 \rightarrow \cdots \rightarrow t_k$[6] of TCG($\lambda$). λ is fully-observable on E up to level k if ρ is a feasible path over E for the test λ.*

3.5.1 Selecting for Addition Changes

Consider an addition change, $\delta = \langle +, t_a\rangle$. The main steps of an incremental procedure to determine if the test λ is a candidate for change δ are given below. The procedure takes the compatibility information among the transitions of the original EFSM, the graph $TCG(\lambda)$, and the added transition t_a as its inputs and returns 1 if λ is a candidate test for δ and returns 0 otherwise. It also outputs the updated compatibility graph to be used for future changes.

[6]For brevity, we ignore the labels on the edges in the path; t_k appears in level k.

1. Update the matching sequence $\phi(\lambda)$ by adding t_a to the appropriate matching sets.

2. Suppose that transition t_a occurs exactly once in the k^{th} matching set of the matching sequence $\phi(\lambda)$. Using the input graph $TCG(\lambda)$ involving the original transitions check if λ is fully-observable on E_1 up to level k. If so, the transitions from E_1 will process the first k test inputs and t_a will not process the k^{th} test input.

   If λ is fully-observable only up to level k - 1 then let formula $F = (L(t_{k-1}) \wedge Emgpos(t_{k-1})) \implies Emgpre(t_a)$ where t_{k-1} is the node marked at the level k - 1 of the input graph. If F is not a valid formula then t_a will not process the k^{th} test input (also the case if λ is fully-observable up to a level less than k - 1). So, λ is not a candidate and the compatibility graph is unchanged.

   If λ is fully-observable up to level k - 1 but not up to level k, and F is a valid formula then λ is a candidate since by Theorem 3, the transition t_a will process the k^{th} test input when λ is applied on E_2. Then, update $TCG(\lambda)$ by adding node t_a at level k and edge (t_{k-1}, t_a, s). Attempt to extend the feasible path to level $k+1$ using the full-observability procedure described in Figure 3.3 on the updated graph. If successful and t_{k+1} is the marked node at this level then add one more edge, (t_a, t_{k+1}, s), to the updated graph and output the resulting graph.

3. If t_a occurs in many matching sets all covered by interval [l, m] and λ is fully-observable on E_1 up to level m or higher then λ is not a candidate and the graph is unchanged. Otherwise, process each matching level in the interval, starting at level l, as described above. Move to the next level only if t_a does not process the test inputs at any of the previous levels.

4. Finally, if t_a does not appear in any of the matching sets of the sequence $\phi(\lambda)$ then the test λ is not a candidate and the graph is unchanged.

The above incremental procedure [75, 90] uses the original compatibility graph to identify candidate tests. The graphs are locally updated so that they can be similarly used in selecting tests for future changes. Such an incremental procedure is likely to be effective in practice since it is bounded by the size of the matching sets affected by the change and is independent of the overall size of the EFSM. Further, focussing on the earliest occurrence can reduce the analysis time, especially for long tests.

Theorem 4 *Test λ is selected for an addition change $\delta = \langle +, t_a \rangle$ if and only if λ exercises δ when λ is applied on the modified EFSM.*

Proof: ($\Rightarrow$:) Suppose λ is selected by the procedure in step 2. Then, $t_a \in T(i_k)$. Also, λ must be fully-observable up to at least level k - 1 but not up to level k on E_1. Since λ is fully-observable up to level k - 1, by Definition 4, there exists a path $start \rightarrow t_1 \rightarrow \cdots \rightarrow t_{k-1}$ in the original TCG(λ) such that the transition sequence $\rho = [t_1, \cdots, t_{k-1}]$ obtained from this path is a feasible path on E_1 from the concrete global state g_0. Hence $Ecpos(\rho, g_0)$ must be satisfiable on E_1, which implies by Lemma 1 that the antecedent of formula F, $L(t_{k-1}) \wedge Emgpos(t_{k-1})$, is satisfiable on E_1. Further, since the formula F must be valid in step 2, it follows by Lemma 1 that $Ecpos([t_1, \cdots, t_{k-1}, t_a], g_0)$ is satisfiable. Therefore, the sequence $[t_1, \cdots, t_{k-1}, t_a]$ must be a feasible path on E_2 from g_0. Hence λ will exercise the change δ when applied to E_2. However, if λ is not selected in step 2, then it must be selected in step 3, which means that λ is fully-observable up to some level m greater than k - 1 then t_a must occur in matching sets following $T(i_k)$ in the sequence $\phi(\lambda)$. We can establish for each of these occurrences that if λ is selected then it will exercise the change. ($\Leftarrow$:) Suppose that λ exercises the change δ on E_2. This implies that there exists a feasible path, say, $\rho = [t_1, \cdots, t_{k-1}, t_a]$ on E_2 from the concrete global state g_0, which means that $Ecpos(\rho, g_0)$ is a satisfiable formula on E_2. Hence by Lemma 1, $g_0 \wedge Emgpos(t_1) \wedge \cdots \wedge Emgpos(t_a)$ must be a satisfiable formula. Now, since the sequence ρ forms a feasible path from g_0, each transition appearing in ρ must process the corresponding test input of λ. Therefore, by Proposition 2, sequence ρ belongs to the matching sequence $\phi(\lambda)$. Hence λ must be fully-observable up to level k - 1 on the original TCG(λ). But λ is not fully-observable up to level k since ρ containing the new transition t_a forms a feasible path, and the change δ preserves determinism. Hence all the conditions of step 2 of the procedure to select λ are satisfied and therefore, the procedure will select λ. $\square$

3.5.2 Selecting for Deletion Changes

Candidate tests for deletions are identified using an incremental procedure similar to that used for addition. The main steps of the incremental procedure are similar to those of the procedure for the addition change described above. The only difference is in the updating of the compatibility graph. While handling a deletion change, if a node corresponding to transition t_d at any of the levels of the graph is marked by the full-observability procedure over E_1 then the corresponding test is chosen as a candidate for the change. In this case, the updated graph is obtained by deleting every node and the resulting dangling edges from the graph corresponding to transition t_d. The process is repeated with all the nodes that do not have an immediate predecessor or an immediate successor until no more such nodes can be found and the resulting graph is returned as the updated compatibility graph.

An important goal for validating a deletion change is to highlight the unanticipated consequences of removal of transition t_d, if any. Therefore, tests that become incomplete over the modified EFSM due to absence of t_d can be potentially important for validating such changes. However, some of these candidate tests may be unusable and need to be discarded as discussed below in Section 3.5.4.

3.5.3 Selecting for Replacement Changes

For a replacement change $\delta = \langle -/+, (t_d, t_a)\rangle$, we can effectively identify candidate tests by analyzing the relation between the transitions t_d and t_a. A replacement change is **local** if t_d is enabled in any global state in E_2[7] then t_a is enabled in that state or vice versa, i.e., the input message and the input state of the transitions are identical and either guard of t_d implies that of t_a or guard of t_a implies that of t_d. In a local replacement, the transitions differ only in their output states and/or their actions.

Candidate tests for local replacements can be identified relatively easily by analyzing only the added or the deleted transition. If the guard of t_d implies that of t_a, we determine if a given test λ is a candidate for the added transition. If λ is not a candidate for addition then it cannot be a candidate for deletion as well and hence λ is not a candidate for the replacement and the compatibility graph remains unchanged. However, if λ is a candidate for the addition then it is chosen to be a candidate for the replacement and the graph is updated as described in the addition procedure above. On the other hand, if the guard of t_a implies that of t_d then repeat the above procedure for deletion only. For non-local

[7]Note that though t_d is not present in E_2 but it can be checked whether or not it is enabled in a global state of E_2.

replacements, we check both added and deleted transitions using their respective procedures. If λ is a candidate for neither then it is not a candidate for the replacement and the graph remains unchanged. Otherwise, λ is chosen to be a candidate and the updated graph is obtained by performing the updates for added transition and/or the deleted transition. Note that though candidate tests can be selected by only considering addition (or deletion) for local replacements, updating of the compatibility graph does require analyzing deletion (or addition).

We have focussed on fully-observable tests in this dissertation, as the descriptions of these tests contain adequate information to efficiently predict that their execution will exercise changes. However, the approach can be generalized to handle tests that are not fully-observable. The failure of the invariant for tests that are not fully-observable can be analyzed to provably predict whether or not these tests will exercise changes. Some of our preliminary work in this direction appears in [89].

3.5.4 Pruning Unusable Tests

A change may make certain tests unusable on the modified EFSM E_2. Tests may become unusable either due to the modification of interface of the original EFSM E_1, or tests not completing on E_2, or test applications on E_2 producing unintended output mismatches.

Usually, the interface of E_2 may change from that of E_1 due to a transition level change that either deletes a message (or state) from the EFSM E_1 altogether, modifies the parameters of an existing message, or adds a new message (or state). A test from the original test suite T whose description refers to an older message (or state) is unusable on the modified EFSM

E_2. Tests becoming unusable on E_2 due to such interface changes are easily identified and removed from the original test suite before identifying candidate tests exercising changes.

Of the remaining tests in T, we consider the tests identified as candidates for regression. Some of these tests could be unusable because either they do not complete or produce unintended output mismatches. Suppose λ is a candidate test which exercises the change while processing its k^{th} test input using the transition t_k, producing the concrete global state g_k. To determine if λ is unusable, we attempt to complete the feasible path by symbolically executing the modified EFSM from the concrete global state g_k by treating the input parameters of each transition as distinct uninterpreted variables. If an initial global state is reached then this means that there exist one or more assignments to the input parameters of the transitions and any of these assignments can be used to extend the path to a feasible run. Each of these assignments constitutes a sequence of test inputs following the input i_k that can be used to complete the test λ. If the original test inputs $i_{k+1}, \cdots, i_n$ of λ belong to these satisfying assignments then λ is selected since it is exercising the change. Otherwise, it is declared unusable and discarded. Test λ is also selected if no satisfying assignments exist, i.e., the test cannot be completed by symbolic execution, because in this case λ is highlighting a potential adverse impact of the change. Note that for a deletion change all tests that do not complete on the modified EFSM will be selected.

Note also that, in general symbolic execution need not terminate when there are loops depending on the input parameters. Such cases are handled by unrolling the loops by a specified threshold value. The values may be set depending on the maximum length of test cases to be included for regression. If the path can be extended to a run by executing the loops up to the specified threshold values then the test is declared usable. Otherwise, the test is selected.

A complete candidate test can be unusable because changes require the test outputs to be modified. Given the modified test purpose, such unusable tests can be identified and removed using test extended concrete post-images and comparing the generated and expected outputs.

Automatically identifying candidate tests in the absence of additional information such as an updated specification is difficult in general. As discussed above, not all incomplete tests or producing unintended output mismatches can be considered unusable. Such a behavior of a test may be caused either because the original test description does not conform to the updated specification or because the formulated change does not do so. Our basic idea to identify unusable tests in the absence of additional information is – declare the test unusable, if the test can be patched to generate a different test whose application completes on the modified EFSM. Otherwise, no such test can be designed for the modified EFSM and hence the formulation of the change is declared to be problematic and the given test can be used to detect the potential fault in the formulation.

3.6 Selecting Multiple Tests

Building a regression test suite by individually analyzing each test in a given test suite can be costly for large test suites. Often, several tests in the given test suite start in the same concrete global state and have overlapping test inputs. For instance, it is typical for tests to use the same inputs to bring the EFSM to a common state and then exercise other specific behaviors. Such tests can be selected (and discarded) simultaneously whenever a given change matches these tests at the overlapping portions of their test inputs. We now describe a procedure that given a test suite and a change, simultaneously selects and discards groups of tests to build a regression test suite.

To simultaneously select and discard tests, the available test suite $T = \{\lambda_1, \cdots, \lambda_n\}$ of the original EFSM E_1 is partitioned into groups of tests. Each group $G = \{\lambda_1, \cdots, \lambda_k\} \subseteq T$ consists of tests that are applied in the same concrete global state g_0 and share a non-empty prefix of their test inputs, i.e., they have at least the same first test input.

Each group G is represented using a test suite tree (TST). Each node of the TST tree denotes a test input occurring at a particular position in the non-empty prefix of the test inputs of the tests in G. The root node of TST denotes test input of the tests in G occurring at the first position of the prefix. Node v is a child of node u in TST if in some test in G, the test input i_u at a position p in the prefix is an immediate predecessor of the test input i_v at position $p + 1$. The edge between a parent node u and child node v is labeled by the set consisting of all tests where this is the case. Further, the set of tests labeling an edge between a parent and a child node is the union of all tests appearing in the subtree rooted at the child node.

Essentially, each TST can be viewed as a trie of test input sequences and the set of tests in TST are refined as we go down the tree.

Example : The TST in Figure 3.5 represents the test suite of Figure 3.1(b). $\Box$

Below, we describe a procedure to simultaneously select and discard tests from T to build a regression test suite T' for the modified EFSM E_2 obtained using the addition change

$\delta = \langle +, t_a \rangle$.

Consider a TST comprised of a group of tests G all starting in the concrete global state g_0. To select tests from this TST, for each node u, the set of transitions of the EFSM E_2 matching the test input i_u, $T(i_u)$, is computed. Each matching set $T(u)$ of the node u is maintained at that node. If the added transition t_a does not appear in any of these matching sets then none of the tests in the TST are chosen as candidate for regression.

Suppose that t_a appears in some matching set $T(i_u)$ at node u of the TST. Let the sequence of matching sets from the root node of the TST to the node u be the matching sequence α =$[T(i_1), \cdots, T(i_n), T(i_u)]$. We build a compatibility graph using the transitions appearing in α and use the procedure in Figure 3.3 with the compatibility graph and g_0 as its inputs to determine if some transition sequence belonging to α forms a feasible path from g_0 on E_2. If the procedure returns *Success* and transition t_a is the transition marked by the procedure in the matching set $T(i_u)$ then all the tests labeling the edge between i_n and i_u in the TST are chosen as candidates. The node i_u and its descendants are removed from TST since all the tests appearing in this subtree are already chosen.

If the procedure returns *Fail* because only a prefix of the matching sequence α, say, $[T(i_1), \cdots, T(i_m)]$ contains a feasible path then the TST is updated by setting the matching sets $T(i_1)$, $\cdots$, $T(i_m)$ to the respective transitions marked by the procedure. Then, the subtree of TST rooted at the node i_{m+1} is removed (all tests in this subtree can be discarded since they are incomplete tests that do not exercise the addition change and are therefore, unusable for the modified EFSM E_2.).

The procedure can also return *Success* but t_a may not be the marked transition in the matching set $T(i_u)$. In this case, we update the matching sets with the transitions marked by the procedure to incrementally continue analyzing the extensions of the matching sequence α reaching other nodes of the TST whose matching sets, if any, include the transition t_a.

The left-right traversal of the updated TST (and the test suite T) is continued until all nodes in the TST whose matching sets contain t_a have been analyzed. All the tests chosen as candidates in the TST are included in the regression test suite T' and the same process is repeated with each of TST.

Example : The matching nodes u for the bank example are highlighted in Figure 3.5. A left-to-right traversal of this tree selects tests λ_2-λ_5 at level 4 after which the nodes at the lower levels can be removed.□

The procedures to select multiple tests for deletion and replacement changes are similar.

3.7 Evaluation

We refer to our approach as SPG (selection with provable guarantees) in this section. We have implemented SPG and applied it several web services, protocols, as well as many model programs taken from a well-known testing benchmark [31]. Our objectives for these experiments are to answer the following research questions.

- RQ1: Can SPG safely and effectively *reduce* the size of the test suite to build regression test suites, whose tests are provably guaranteed to exercise a change to system?
- RQ2: Can SPG *reduce* a substantial amount of time to rerun the test suite?

Moreover, we compare SPG precision and inclusiveness with those of the earlier dependency based approaches (DEP) from [20,61,62,79], and with our own brute-force symbolic execution approach (SYM). We also study the effectiveness of the approach in selecting tests that can detect certain faults that can be mapped back to changes over the models.

3.7.1 Experimental Design

Our prototype is coded in Perl and C on a Linux server with 4GB memory and employs built-in graph libraries. It uses the reasoning framework, *SAIL*, implemented based on the theorem prover *Simplify* [29] extended with queues [88]. *SAIL* is used in a push-button manner to discharge proof obligations. The major activities for setting up our experiments are as follows.

Change Generation. Changes to EFSMs play an important role in all of our applications. We use the changes provided by the applications whenever they are available. In addition, changes are synthetically generated using the following simple process. Given an EFSM model (text files), the number of transitions to be changed, and the overall number of modified EFSMs, as inputs, the input EFSM is first compiled into a graph[8] after ensuring its well-formedness, determinism, and consistency [87]. A modified graph corresponding to a modified EFSM is built from the input graph by marking the number of transitions given as input with the change actions a (addition) or d (deletion), chosen randomly. The process is repeated to generate the number of modified EFSMs specified as input.

Note that for single transition changes, a modified EFSM corresponding to each transition and each change action is produced. Consequently, for these changes, occurrences

[8]For examples of the graphs, please see the EFSMs of the particular applications depicted later in this section.

of transitions matching changes are uniformly distributed over the inputs of each test in the test suite.

Test Generation. The test suite for the original EFSM is hand crafted wherever possible, such as those for model programs from [31]. Tests were also automatically generated using the model-based test generator ParTeg [99]. The current version of ParTeg has limited support for EFSMs containing self-loops and aggregate data types such as arrays. To perform test generation for such EFSMs, first, the given EFSM E is changed to an EFSM E' as follows – 1. the self-loops in E are unrolled a specified threshold number of times. 2. Array accesses are disambiguated using their most recent instances and then, each distinct array access is replaced by a distinct new integer variable. Tests are generated based on E' using ParTeg. The values assigned by ParTeg to the new integer variable are then mapped back to the array accesses to construct a test having an array input value. Generated tests traverse the self-loops in E up to the threshold number of times.

Note that it may not always be feasible to form an array value by using the integer input values assigned by ParTeg. For instance, semantically equivalent but lexically different array accesses such as $B[i + j]$ and $B[j + i]$ are replaced by different integer variables and may be assigned different input values by ParTeg in which case no array value can be built. We weed out such unusable tests and others by using the theorem prover. The remaining tests are then added to the hand-crafted tests to form a test suite for the original EFSM. The changed EFSM E' used for test generation is discarded after generating tests. Further, the change generation and the test generation are totally independent of each other. Hence the above test generation process has limited impact on our experimental results.

Selection with provable guarantees (SPG). To perform regression test selection using the SPG approach, compatibility information for the transitions appearing in the test suite is computed, compatibility graphs are built, and a regression test suite for each change is produced using the original and modified EFSM graphs as described in Section 3.5 and Section 3.6. Efficiency, precision, and inclusiveness of the approach are computed based on the costs [78] – C_1(retest-all) the cost to re-run the entire original test suite[9]; C_2 (selective-retest) the cost of running the regression test suite; and C_3 (regression analysis) the cost to build the regression test suite.

Symbolic Execution (SYM). To study the benefits of the selective use of the reasoning engine and the incremental procedures in the SPG approach, we adapted symbolic execution [59] to apply tests on EFSMs. Our implementation instruments the original and/or modified EFSM graphs with the transitions appearing in the changes and applies a test by repeatedly computing concrete post-images over these graphs until either one of the instrumented transitions is executed or a global initial state is reached, or the procedure cannot make progress. Tests executing an instrumented transition are selected for regression. In the SYM approach, in each step, every EFSM transition is considered, a post-image is formulated, and its satisfiability is checked using the reasoning engine. The path is extended using the transition if the corresponding post-image is satisfiable.

Dependency based EFSM Selection (DEP). We also implemented the DEP approach described by Korel *et al* in [62]. Since tests are viewed as sequences of transitions ignoring input values in [62], matching sequence for each original SPG test is computed by using the appropriate EFSM graphs and each transition sequence belonging to the matching sequence

[9]Tests that become unusable due to interface changes can be detected at a negligible cost and are removed.

of the SPG test forms a DEP test. A DEP test is a regression candidate if the transitions in the change appear in that test. Candidate DEP tests causing identical dependency patterns[10] in the modified (or original) EFSM control and data dependency graphs are equivalent and used to remove redundant candidate DEP tests. The remaining DEP tests are mapped back to the original SPG tests to build a regression test suite as follows. Select an original SPG test only if some DEP test derived from this test causes a dependency pattern that is not identical to the dependency pattern caused by any of the other DEP tests. The analysis time C_3, inclusiveness, and precision are computed for DEP approach in terms of the SPG tests and not DEP tests. However, C_3 does not include the time to map SPG tests to DEP tests and vice versa.

3.7.2 Case Studies

We have applied the prototype to ten examples from the literature: completion (*Cmp*), two-phase commit (*Tcp*), and conference (*Cnf*), and third-party call (*Thp*), Cruise Control (*Con*), Printtokens (*Pri*)[11], automatic teller machine (*Atm*) [20, 62], bank (*Bnk*), vending machine (*Ven*), and a Microwave oven (*Mic*) [22]. The completion, two-phase commit, and conference protocols described on the web-site [12] have been used earlier to evaluate formal testing approaches. The completion protocol *Cmp* is used by an application to tell the coordinator to either try, commit, or abort an atomic transaction. The two-phase commit *Tcp* is a coordination protocol that defines how multiple participants reach an agreement on the outcome of an atomic transaction. The conference protocol, *Cnf*, is a chatbox-like

[10]Please see Section 2.2 for more details.
[11]Software-artifact Infrastructure Repository: http://sir.unl.edu/portal/index.html
[12]http://fmt.cs.utwente.nl/ConfCase/

protocol. The EFSM models for *Cmp* and *Tcp* are manually created using their graphical and textual descriptions and contain 7 and 14 transitions respectively. Their test suites have 300 and 800 tests respectively. For the *Cnf* protocol, we used the EFSM description (c) available from the web site referred above. This model has 19 transitions and the test suite has 723 tests. The web site gives two EFSMs called (c) and (d) and describes four changes to transform EFSM (c) to EFSM (d). The four changes specified there are all additions that allow members to send data before joining the conference. The third-party call (*Thp*) is a protocol from Praxis, whose EFSM has 15 transitions and test suite has 837 tests [56].

Cruise Control (*Con*) and Printtokens (*Pri*) are programs from the popular test benchmark [31]. These programs are manually translated to obtain the EFSM models. The EFSM for *Con* has 13 transitions and its test suite has 1000 tests. The EFSM for the program *Pri* is omitted in the dissertation due to its size. We briefly describe how this EFSM was created from the program. The C code for *Pri*, given Appendix, reads an input file containing strings delimited by white space, terminated by ";", and tokenizes them into categories such as identifers, characters, numbers and so on. Based on the input symbol read and the current token category, a switch statement in the code determines the next token category and processes the input symbol. The current token category is output if the input symbol is a whitespace. Each case of the switch statement is mapped to an EFSM transition whose input message is the input symbol read, input state corresponds to the current category, and the output state is the next category. The actions and the output message of the transition are obtained by translating the case statement body. Each test file provided with the *Pri* application is translated into an EFSM test whose individual test inputs correspond to the string elements and the outputs are the expected token values.

Microwave Oven (*Mic*) is originally described as a Kripke structure [22]. We simply modified the labels on the arcs to obtain EFSM transitions by adding input and output messages. The model has 12 transitions and 1160 tests. The remaining examples web automatic teller machine (*Atm*) (6 transitions and 800 tests), bank (*Bnk*) (9 transitions and 1124 tests), a vending machine example (*Ven*) (8 transitions and 87 tests) all appear as EFSMs in earlier testing literature [91] and were used as such.

3.7.3 Study Results and Discussion

Our results for the SPG approach are summarized in a table in Table 3.1. The first column of the table lists the ten examples along with the number of EFSM transitions. The second column lists the test suite size and the average test length for each example. The third column lists the total number of changes made to each example. Columns four, five, and six show C_1, the cost for running the full test suite, C_2, the cost for running the selected tests, and C_3, the cost for performing analysis respectively. These columns list the average costs per change. Column seven lists the average number of unusable tests per change for each example. Next column is the average number of selected tests per change. Finally, the last column lists the average time savings per change, defined as the percentage $(C_1 - (C_2 + C_3))/C_1$, based on the cost model of [68]. As seen from Table 3.1, SPG achieves an average time savings of around 30% while achieving an average reduction of around 40% in the size of the test suite selected for all attempted examples. Non-zero number of unusable tests are identified in all of the examples with up to nearly 15% of tests being unusable in some cases. First, we describe these three aspects in detail below. Then, we discuss the effectiveness of SPG approach for detecting faults.

Table 3.1: SPG Regression Test Selection Costs Table

Example	**Test Suite**		**Number of changes**	**C1 (Secs)**	**C2 (Secs)**	**C3 (Secs)**	**Average number of unusable tests**	**Average number of selected tests**	**Average time savings (%)**
	Size	Average length							
Con (13)	1000	78	26	2152.31	837.34	629.39	31	386	31.9
Prn (99)	1439	102	3	6345.04	3511.73	2035.87	62	798	12.6
Atm (6)	800	50	12	1210.13	842.97	214.63	89	486	12.6
Mic (12)	1160	12	24	289.2	92.3	7.45	25	614	65.5
Bnk (9)	1124	35	18	2483.11	738.96	1041.34	43	364	28.3
Thp (15)	837	66	30	1249.41	367.91	232.11	58	203	52
Ven (8)	87	37	16	92.38	23.13	50.35	12	20	20.4
Cnf (19)	723	47	38	629.23	328.13	187.43	32	257	18.1
Cmp (7)	800	59	14	532.21	252.66	132.71	12	316	27.6
Tcp (14)	311	18	28	147.47	48	27.76	39	102	48.6

RQ1. Reduction in Test Suites and Unusable Tests? The variance in the number of selected tests largely depends on the number of transitions in the EFSM models, and those appearing in loops. Consider the examples *Atm* with 6 transitions and high (486) average number of selected tests and *Ven* with 8 transitions and a very low (20) average number of selected tests. The difference in the average number of selected tests in these examples can be attributed to the number of transitions appearing in loops. Almost all transitions of example *Atm* appear together in one or more loops. Hence a feasible run corresponding to each test is likely to contain all of the transitions and this leads to a high number of selected tests. This is in contrast with *Ven* where loops involve at most one or two transitions.

As expected, reduction in test suite size is directly correlated to the time savings in examples such as *Prn*, *Atm*, and *Thp*. However, it is not so in *Con*, where almost 70% of the original test suite is eliminated but the time savings are not as much. On further examination of the discarded tests we concluded that these do not take much time to run.

Similarly, we observed in example *Mic* that few tests are eliminated but the time savings are higher.

The varying number of unusable tests found in these examples depends largely on the interaction among transitions. Usually, examples such as *Atm*, *Bnk*, and *Ven* whose transitions with complex guards interact with each other in subtle ways seem to produce more unusable tests. We can also reduce the number of unusable tests by using carefully hand-crafted tests.

RQ2. Time Savings? Varying amount of time savings obtained across these examples can be mainly attributed to three reasons – complexity of the data values, and data types used in the EFSMs and the tests, exploiting overlap in the test descriptions, and the compatibility of transitions. Time savings higher than 50% for *Mic* can be mainly attributed to simple data values and data types such as boolean and integers whereas *Atm* and *Bnk* do not have as much time savings since their EFSMs and tests involve complex aggregate data types such as arrays. We believe that more savings can be realized for examples with aggregate data types with the ongoing advances in the SMT solvers [1,2]. Time savings are significant for *Thp* because the input messages in its EFSM have no parameters. Consequently, its tests do not involve any concrete values and allow for lot more overlaps in the test descriptions. These overlaps are effectively exploited by our simultaneous test selection procedures using the TST trees. Compatibility of transitions played an important role in all examples in obtaining time savings.

Fault Detection Using SPG. We also studied whether faults in the system under test can be detected by SPG. We considered faults that are caused solely due to the changes in the model. The criterion for test selection used by SPG is a necessary condition for detecting such faults. We used the popular TCAS example from the test benchmark [31] with 41 versions and 1590 tests. We chose the faulty versions *Ver1*, *Ver2*, *Ver6*, *Ver7*, *Ver8*, and *Ver9* produced by mutation analysis. We created models from each of the faulty versions of the code and translated the code-based tests to model-based tests. SPG was used to select the model-based tests. The corresponding code-based tests were run on the original and modified code to identify the model-based tests revealing faults. Our results are depicted in Table 3.2. The first column gives the faulty versions; the second column gives the tests selected by SPG; the third column gives the number of selected tests that reveal faults. As shown, SPG was able to identify a non-zero number of fault-revealing tests for each version. To further check if SPG missed any of the fault-revealing tests, we ran all the code-based tests in the original test suite on both the original and modified C programs and collected the tests producing different outputs. We compared the corresponding model-based tests and those identified using SPG and found these two sets of tests to be identical in all cases. Hence SPG is able to select all the fault-revealing tests in all the versions in this example, which is quite encouraging.

Table 3.2: Fault Detection for TCAS

TCAS (1590 tests)	**Number of selected tests**	**Number of fault revealing tests**
Ver1	432	130
Ver2	527	61
Ver6	331	12
Ver7	1560	36
Ver8	1560	1
Ver9	508	9

Multiple Changes. To study the effect of multiple changes, several modified models were generated by changing up to 10 transitions in each example. SPG was applied on each of these modified models. The average time savings obtained for each example is depicted in Figure 3.6. The X-axis plots the number of changes made to each example and the Y-axis plots the average time savings obtained. As expected, our results show that for every example, the time savings reduce as the number of changed transitions increases. The time savings reach zero in examples such as *Atm*, *Cmp*, and *Ven* that have less than 10 transitions since the entire model is eventually changed and hence all the tests in the original test suite have to be selected. In Figure 3.7, the X-axis plots the number of changes made to each example and the Y-axis plots the number of selected tests. The results show that the average number of selected tests increases with the increased number of transition changes, which is not surprising. For examples like *Con*, and *Cnf* there is sometimes a sharp increase in the average number of selected tests because the newly changed transitions dominate other transitions and hence appear in more test runs.

SPG vs. SYM. The first and third bars in Figure 3.8 depict the results of our comparison of the analysis costs (C_3) of the SPG and the SYM approaches. The X-axis plots the examples and the Y-axis plots the analysis cost in seconds. The results show that the cost of using SYM are much higher than that of SPG for all examples. We attribute this to two factors. First, SYM does not exploit the test description and hence analyzes every EFSM transition in each execution step. Second, SYM has to analyze non-modification traversing tests in their entirety. The inclusiveness and precision of SYM are the same as that of SPG for modification traversing tests.

SPG vs. DEP. We compared SPG and the DEP approaches in terms of the analysis cost (C_3) and the number of selected tests. Our results are depicted in figure 3.8 and figure 3.9. In figure 3.8 the second and third bars depict the analysis costs for DEP and SPG respectively. As seen from this figure, the analysis cost for DEP is higher than that of SPG in all these examples. We attribute this to DEP having to consider all of the exponential transition sequences belonging to the matching sequence of a test and compare their dependency patterns to find the sequences that are to be selected. As explained before, SPG analysis costs vary based on the characteristics of the examples and this leads to varying analysis cost differences between the SPG and DEP approaches shown in figure 3.8.

The figure 3.9 depicts the number of tests selected by SPG and DEP for all examples. As seen from this figure, there is no direct correlation between the two approaches for the number of tests selected in these examples. Based on a closer analysis, we observe that DEP selects more tests mainly because of two reasons – *i)* does not discard unusable tests, *ii)* chooses valid tests that do not exercise a change. Such tests are discarded by SPG. We observed that DEP selects more tests in examples *Atm*, *Bnk*, and *Cnf* because of choosing unusable tests. More tests are selected by DEP for examples *Prn* and *Mic* because it cannot distinguish among valid tests that do not exercise the change.

More specifically, consider tests λ_4 and λ_5 in the original test suite in Figure 1(b). Both of these tests are unusable tests and this cannot be discerned based on their dependency patterns. Further, DEP cannot distinguish the dependency patterns of these two tests with respect to the change transition t'_{12}. Hence DEP will select one of these tests at random whereas both these tests will be discarded by SPG. As another example, consider the test λ_6 = [*Open*(100)/*ack*(1), *deposit*(1, 25)/*ack*(*B*[1]), *wdraw*(1, 310)/*ack*(*B*[1]), *wdraw*(1, 10)/*ack*(*B*[1]), *close*(1)/*ack*(*B*[1])] along with the other 5 tests the original test suite in

Figure 1(b). Test λ_1 and λ_6 are both valid tests for the modified EFSM but neither exercises the change. However, DEP will select one of these tests because it cannot distinguish them based on dependency patterns whereas SPG will discard both.

We observed that DEP selects fewer tests in the examples *Con*, *Thp*, *Ven*, *Cmp*, and *Tcp* when the dependency patterns are identical for tests exercising changes and those not exercising changes. More specifically, consider a test λ_7 = $Open$(100)/ack(1), $deposit$(1, 25)/ack(B[1]), $wdraw$(1,60)/ack(B[1]), $wdraw$(1,10)/ack(B[1]), $close$(1)/ack(B[1]) along with the other 5 tests in the original test suite in Figure 1(b). Tests λ_1, λ_2, and λ_7 potentially execute the change transition t'_{12} in Figure 1(b) since they all have $wdraw$ messages that is an instance of t'_{12}'s input message. However, λ_1 does not exercise the change whereas the other two tests do. Only one of these tests is selected by DEP since it cannot distinguish them based on their dependency patterns whereas SPG selects the tests λ_2 and λ_7. Therefore, DEP selects fewer tests because it does not select a test exercising the change.

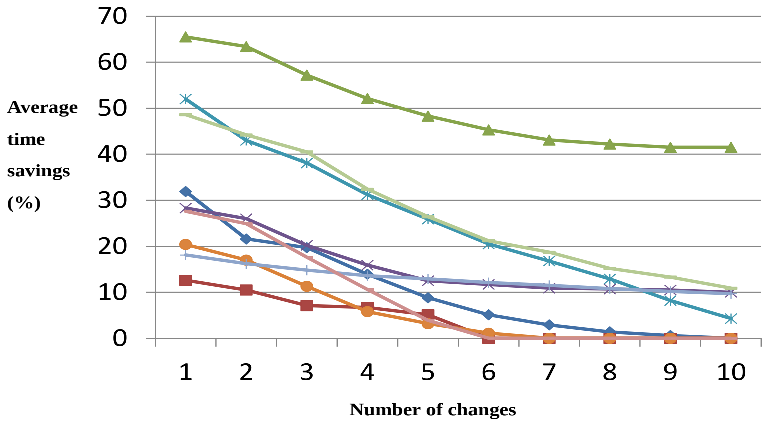


Figure 3.6: Time Savings across multiple changes

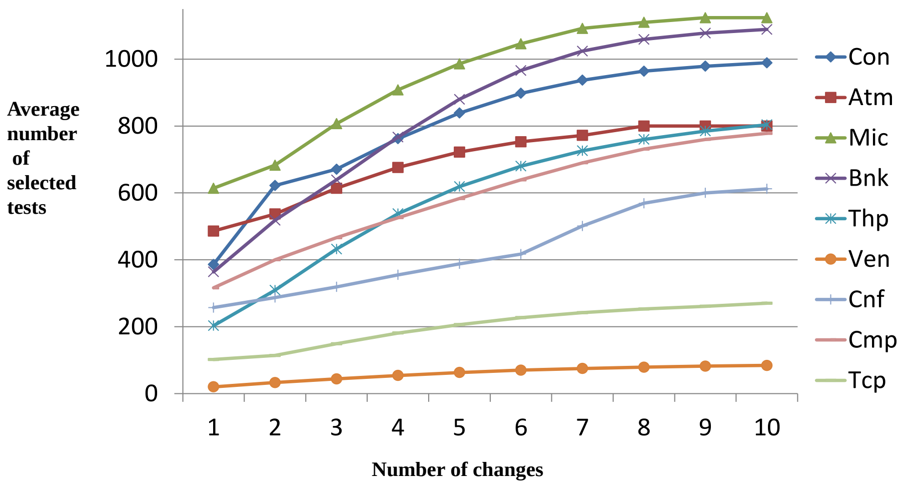


Figure 3.7: Tests Selected across multiple changes

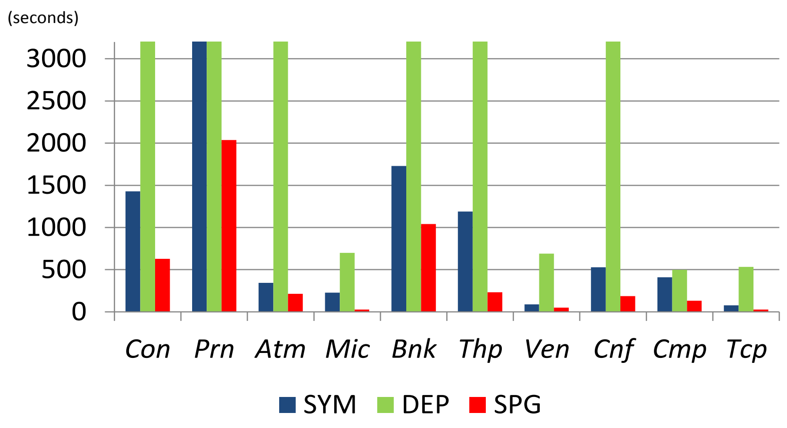


Figure 3.8: Analysis Costs of SYM, DEP and SPG

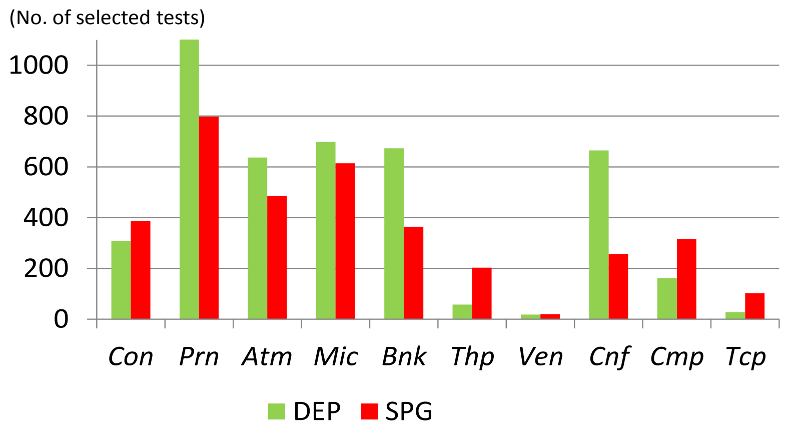


Figure 3.9: Selected Tests of DEP and SPG

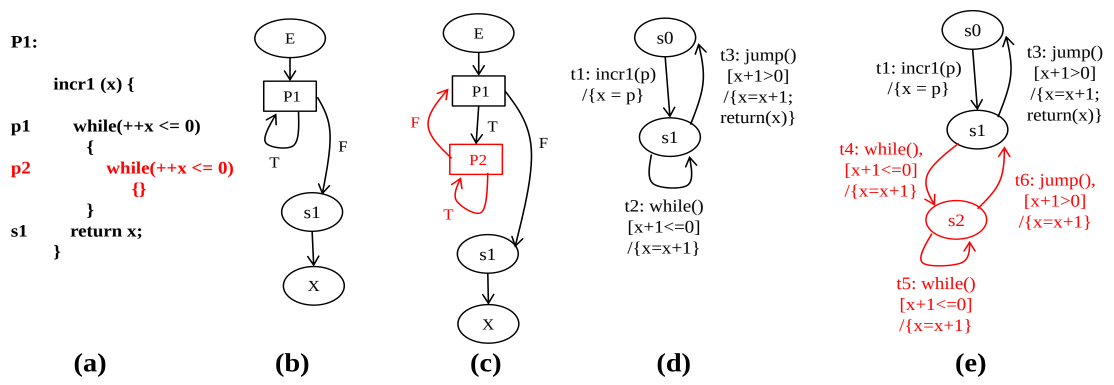
3.7.4 SPG and Code-based Approaches

We now compare SPG with the well-known graph-walk approaches of Rothermal and Harrold [79, 105]. Graph-walk approaches compare original and modified control flow graphs (CFGs) or program dependence graphs (PDGs) derived from original and modified code respectively to determine whether or not a test applied on the original code should be selected for the modified code. A walk of the original and the modified graphs is performed while lexicographically comparing the contents of the corresponding nodes. If two corresponding immediate successors of two equivalent nodes differ lexicographically then the edge in the modified graph is supposed to be exercising a change. In this case, the edge in the original graph is checked for inclusion in the available test executions and all the tests whose executions include this edge are selected for regression.

We compared SPG with the graph-walk approach based on CFGs using an example derived from [79](page 184). The code segment and its CFG are depicted in Figure 3.10 (a), (b) and (c); modified portions are colored red. Consider a test $x = 0$ used on the original code. This test does not exercise the change but will be selected by the CFG (as explained in [79] itself, page 184). This is because the nodes P_1 and P_2 are lexicographically equivalent whereas the immediate successor s_1 of P_1 is lexicographically different from the immediate successor P_1 of P_2 and the edge (P_1, s_1) in the original CFG appears in the execution of the test. For comparison, we first created an EFSM model from the original code, mapped the code changes to EFSM changes, and created a modified EFSM model, as depicted in Figure 3.10(d) and (e). The test $x = 0$, was then mapped to the EFSM test λ: [$incr1(0)$/null, $jump()/return(0)$]. It is easy to verify that SPG will not select this λ since it does not exercise any of the changed transitions t_2, t_4, t_5, and t_6.

Note that the same result can be obtained for the above example, by performing the comparison in the forward direction, i.e., starting with the EFSMs, deriving the corresponding flow-graphs from these models, and applying the graph-walk on the corresponding flow-graphs. We also considered the graph-walk approach based on PDGs and can similarly show that the graph-walk approach based on PDG will select tests not exercising changes whereas these tests are discarded by SPG.

In general, the graph-walk method selects a superset of modification-traversing tests and hence will include all the tests selected by the SPG. This is perhaps not surprising since the graph-walk method is based on lexicographic comparison that may not be able to distinguish control and data paths despite having test executions, whereas SPG is able to differentiate these paths due to the more detailed semantic analysis performed using a theorem prover. Additionally, SPG performs analysis without using prior test executions whereas the graph-walk method requires such executions to be available.



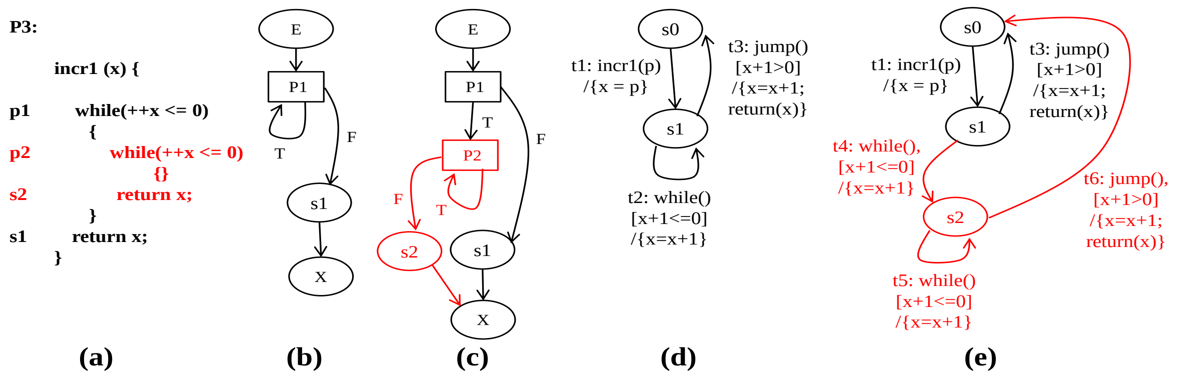
```
P1:

      incr1 (x) {

p1       while(++x <= 0)
         {
p2           while(++x <= 0)
               {}
         }
s1       return x;
      }
```

Figure 3.10: SPG vs. CFG (1) (a)code, (b)CFG, (c)new CFG (d) EFSM (e)new EFSM

In some cases, CFG approach possibly misses selecting tests that are modification-traversing. For example, the code segment and its CFG are depicted in Figure 3.11 (a), (b) and (c); modified portions are colored red. Consider a test $x = -1$ used on the original code. This test does exercise the change but won't be selected by the CFG. This is because each nodes in original CFG and modified CFG are lexicographically equivalent when traversing the graphs. For comparison, we first created an EFSM model from the original code, mapped the code changes to EFSM changes, and created a modified EFSM model, as depicted in Figure 3.11(d) and (e). The test $x = -1$, was then mapped to the EFSM test λ: $[incr1(-1)/\text{null}, while()/\text{null}, jump()/return(0)]$. It is easy to verify that SPG will select this λ since it exercises the changed transitions t_4, and t_6.



```
P3:
      incr1 (x) {
p1        while(++x <= 0)
           {
p2             while(++x <= 0)
                 {}
s2              return x;
           }
s1        return x;
      }
```

E
P1
T
F
s1
X
E
P1
T
F
P2
F
T
s2
s1
X
s0
t1: incr1(p) /{x = p}
t3: jump() [x+1>0] /{x=x+1; return(x)}
s1
t2: while() [x+1<=0] /{x=x+1}
s0
t1: incr1(p) /{x = p}
t3: jump() [x+1>0] /{x=x+1; return(x)}
s1
t4: while(), [x+1<=0] /{x=x+1}
s2
t6: jump(), [x+1>0] /{x=x+1; return(x)}
t5: while() [x+1<=0] /{x=x+1}
(a)
(b)
(c)
(d)
(e)

Figure 3.11: SPG vs. CFG (2) (a)code, (b)CFG, (c)new CFG (d) EFSM (e)new EFSM

The second is to use Program Dependency Graph (PDG) to select tests [77]. In certain case, it is also possible for PDG to select tests that are not modification-traversing by adding new branches. For example, the code segment and its CFG are depicted in Figure 3.12 (a), (b) and (c); modified portions are colored red. Consider a test $x = 1$ used on the original code. This test does not exercise the change but will be selected by the PDG. This is because the region R_1 colored by red in original PDG of Figure 3.12 (b) is affected by adding new branch in the modified program. For comparison, we first created an EFSM model from the original code, mapped the code changes to EFSM changes, and created a modified EFSM model, as depicted in Figure 3.12(d) and (e). The test $x = 1$, was then mapped to the EFSM test λ: [$incr1$(1)/null, if()/null, $jump$()/$return$(2)]. It is easy to verify that SPG will not select this λ since it exercises neither of the changed transitions t_5, and t_6.

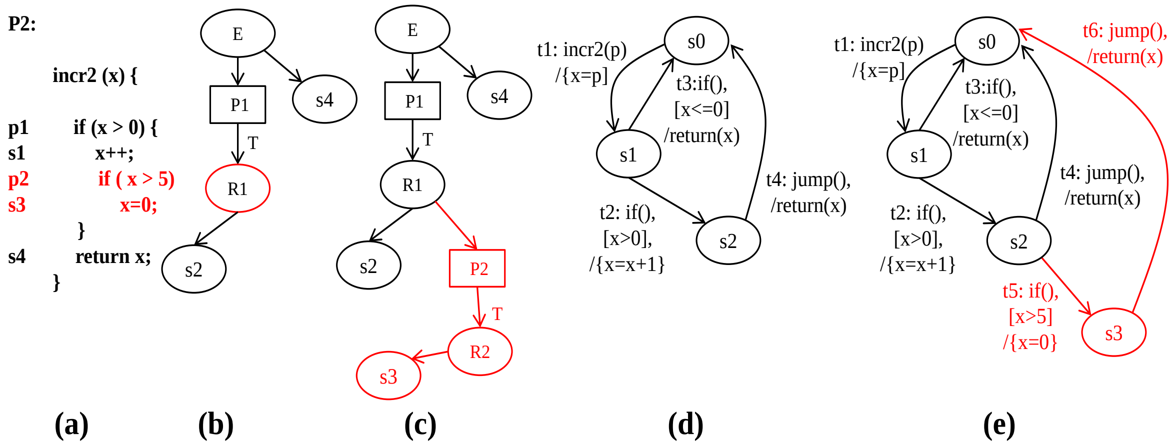


Figure 3.12: SPG vs. PDG (1) (a)code, (b)PDG, (c)new PDG (d) EFSM (e)new EFSM

3.7.5 Threats to Validity

The primary threat to validity of our experiments involves the change and test generation processes. In many examples, we have synthetically generated changes. This is alleviated by generating all changes expressible in terms of additions, deletions, and replacement of transitions. More experiments with real-world data would also be highly beneficial in further demonstrating the effectiveness of the approach. Concrete data assignment in generated tests also influences our experimental results, which was mitigated by use of a theorem prover to weed out inapplicable randomly generated values. Another related issue is the use of multiple different test generators to produce quality test suites. Finally, the authors' subjectivity and bias during manual conversion from the code to EFSMs in some examples can influence our results.

CHAPTER 4

DECOMPOSING COMPOSITE CHANGES to SUPPORT CODE REVIEW

4.1 Motivating Example

Figure 4.1 shows our motivating example from JFreeChart project, an information visualization library to display graphs and charts. We adapt and simplify the example for the presentation purpose. Suppose David updates a program with two independent development tasks: (1) refactorings by standard refactoring techniques [37] and (2) bug-fixes to resolve some graphical rendering issues. First, he applies Pull Up Field to field `paintList` for moving the field to a super class `AbstractRenderer`, and adds a call to a new method `getSeriesPaint`. He then performs Extract Method on three methods `lookupSeriesPaint`, `lookupSeriesFillPaint`, and `lookupSeriesOutlineStroke` by removing reusable code fragments and adding a call to each new method `setSeriesPaint`, `setSeriesFillPaint`, and `setSeriesStroke`, respectively. Second, he fixes a bug by adding a null checker to determine whether field `basePaint` is accessed before its initialization. David, then, commits his changes in one single transaction to a version control repository with a message, *"Updated several methods by applying the refactorings. Also, fixed graphical rendering* *bugs by adding null checkers."*. The commit log message indicates that his changes may be a composite change across files, which requires inspecting line level differences file by file.

Suppose that Monica investigates the code changes made by David during peer code reviews. Since David's modification includes multiple independent code changes, she must inspect his changes while determining which part is related to refactorings or bug-fixes. She needs to inspect line level differences file by file. Recent studies find that code reviewers are most able to understand cohesive changes—strongly related modification between code fragments [76, 92].

To alleviate the burden of inspecting a composite change, ChgCutter allows Monica to interactively select a sub-region of David's changes, and it automatically identifies related changes based on a user-selected change region. It then decomposes a composite change into a set of related changes, and helps her efficiently validate change subsets by applying a regression test selection technique.

Composite Change Decomposition. Firstly, Monica may need to inspect refactorings conducted by Extract Method in method `lookupSeriesPaint`. She may wonder whether there exist other locations that are modified similarly to the refactorings applied to the method `lookupSeriesPaint`. Monica uses ChgCutter as a plug-in built atop Eclipse IDE[1] to select the refactored region in Figure 4.1 (green portions) within Eclipse's *Compare* editor.[2] Given the selected change, ChgCutter automatically generates an *edit matching* template that consists of the differences between the original and edited versions of the program and dependent contexts. The edit matching template is used to find other related,

[1]`http://eclipse.org/pde`
[2]http://googl/MLkGi6

similar locations, such as refactorings performed in methods `lookupSeriesFillPaint` and `lookupSeriesOutlineStroke`, respectively. ChgCutter, then, decomposes the composite change and produces a set of related refactoring changes, excluding non-related changes such as bug-fixes. It also allows Monica to select a different type of changes such as bug-fixes (yellow portions). It then produces another set of related changes, excluding refactorings.

```
  class RendererImpl extends AbstractRenderer {
    public Paint lookupSeriesPaint(int series) {
-       Paint seriesPaint = this.paintList.getPaint(series);
+       Paint seriesPaint = getSeriesPaint(series);
        if (seriesPaint == null && this.autoPopulateSeriesPaint) {
-           DrawingSupplier supplier = getDrawingSupplier();
-           if (supplier != null) {
-               seriesPaint = supplier.getNextPaint();
-               this.paintList.setPaint(series, seriesPaint);
-           }
-           refresh(this.basePaint);
+           setSeriesPaint(series, seriesPaint, false);
+           if(this.basePaint == null)
+               throw new Exception("Null 'basePaint'.");
+           else
+               refresh(this.basePaint);
        }
        ...
    }

+   public void setSeriesPaint(int series, Paint paint, boolean notify) {
+       DrawingSupplier supplier = getDrawingSupplier();
+       if (supplier != null) {
+         paint = supplier.getNextPaint();
+           this.paintList.setPaint(series, paint);
+           if (notify)
+               fireChangeEvent();
+       }
+   }
  }
```

(a) User-selection portions highlighted for these two different types of changes.

```
  class RendererImpl extends AbstractRenderer {
    public Paint lookupSeriesFillPaint(int series) {
-       Paint seriesFillPaint = this.fillPaintList.getPaint(series);
+       Paint seriesFillPaint = getSeriesFillPaint(series);
        if (seriesFillPaint == null && this.autoPopulateSeriesFillPaint) {
-           DrawingSupplier supplier = getDrawingSupplier();
-           if (supplier != null) {
-               seriesFillPaint = supplier.getNextFillPaint();
-               this.fillPaintList.setPaint(series, seriesFillPaint);
-           }
-           refresh(this.baseFilllPaint);
+           setSeriesFillPaint(series, seriesFillPaint, false);
+           if(this.baseFailPaint == null)
+               throw new Exception("Null 'baseFilllPaint'.");
+           else
+               refresh(this.baseFilllPaint);
        }
        ...
    }

+   public void setSeriesFillPaint(int series, Paint paint, boolean notify) {
+       DrawingSupplier supplier = getDrawingSupplier();
+       if (supplier != null) {
+         paint = supplier.getNextFillPaint();
+           this.fillPaintList.setPaint(series, paint);
+           if (notify)
+               fireChangeEvent();
+       }
+   }
  }
```

(b) One changes that are similarly edited for these two different types of changes.

```
  class RendererImpl extends AbstractRenderer {
    public Stroke lookupSeriesOutlineStroke(int series) {
-       Stroke result = this.outlineStrokeList.getStroke(series);
+       Stroke result = getSeriesOutlineStroke(series);
        if (result == null && this.autoPopulateSeriesOutlineStroke) {
-           DrawingSupplier supplier = getDrawingSupplier();
-           if (supplier != null) {
-               result = supplier.getNextOutlineStroke();
-               this.outlineStrokeList.setStroke(series, result);
-           }
-           refresh(this.baseOutlineStroke);
+           setSeriesOutlineStroke(series, result, false);
+           if(this.baseOutlineStroke == null)
+             throw new Exception("Null 'baseOutlineStroke'.");
+           else
+             refresh(this.baseOutlineStroke);
        }
        ...
    }

+   public void setSeriesOutlineStroke(int series, Stroke stroke, boolean notify) {
+       DrawingSupplier supplier = getDrawingSupplier();
+       if (supplier != null) {
+         stroke = supplier.getNextOutlineStroke();
+           this.outlineStrokeList.setStroke(series, stroke);
+           if (notify)
+               fireChangeEvent();
+       }
+   }
  }
```

(c) Another changes that are similarly edited for these two different types of changes.

Figure 4.1: A composite code change example, including refactorings and bug-fixes. Added code is marked with ‘+’, and deleted code marked with ‘-’.

Intermediate Version Construction. Secondly, Monica may need to test David's refactorings to determine that the other unchanged parts have not been adversely influenced by his refactoring edits. ChgCutter constructs an intermediate version of the program (Figure 4.2 left) by automatically applying isolated refactoring edits to the original version. Automated *recompilation* and *generation* for an intermediate version includes critical challenges; the generated program must be *compilable* and *executable*, when exercising the applied changes against tests during the regression testing. To satisfy the requirements, ChgCutter analyzes dependencies on changed entities and surrounding contexts to extract prerequisite edits. Similarly, for testing bug-fixes, ChgCutter constructs another intermediate version (Figure 4.2 right), when she needs to reveal a fault in bug-fixes.

Regression Test Selection. Lastly, Monica can reuse the test suite that was used to test the original version of the program but may need to run an appropriate subset of the test suite to validate an intermediate version. To increase effectiveness of a testing, ChgCutter applies a regression test selection technique to automatically select a subset of the test suit affected by the changes applied to an intermediate version shown in Figure 4.2.

```
class RendererImpl extends AbstractRenderer {
  public Paint lookupSeriesPaint(int series) {
     Paint seriesPaint = getSeriesPaint(series);
     if (seriesPaint == null && this.
   autoPopulateSeriesPaint) {

   setSeriesPaint(series, seriesPaint, false);
         refresh(this.basePaint);
     }
     System.out.println("base is used if series is
   null");
     if (seriesPaint == null) {
         seriesPaint = this.basePaint;
     }
     return seriesPaint;
  }

  public Paint lookupSeriesFillPaint(int series) {

   Paint seriesFillPaint = getSeriesFillPaint(series);
     if (seriesFillPaint == null && this.
   autoPopulateSeriesFillPaint) {

   setSeriesFillPaint(series, seriesFillPaint, false);
         refresh(this.baseFillPaint);
     }
     if (seriesFillPaint == null) {
        seriesFillPaint = this.baseFillPaint;
     }
     return seriesFillPaint;
  }

  public Stroke lookupSeriesOutlineStroke(int
   series) {

   Stroke result = getSeriesOutlineStroke(series);
     if (result == null && this.
   autoPopulateSeriesOutlineStroke) {

   setSeriesOutlineStroke(series, result, false);
         refresh(this.baseOutlineStroke);
     }
     if (result == null) {
        result = this.baseOutlineStroke;
     }
     return result;
 }


   public void setSeriesPaint(int series, Paint paint,
                              boolean notify) {

   DrawingSupplier supplier = getDrawingSupplier();
     if (supplier != null) {
        paint = supplier.getNextPaint();
         this.paintList.setPaint(series, paint);
         if (notify)
             fireChangeEvent();
     }
  }

public void setSeriesFillPaint(int series,

   Paint paint, boolean notify) {

   DrawingSupplier supplier = getDrawingSupplier();
     if (supplier != null) {
        paint = supplier.getNextFillPaint();

   this.fillPaintList.setPaint(series, paint);
         if (notify)
             fireChangeEvent();
     }
 }

 public void setSeriesOutlineStroke(int series,

   Stroke stroke, boolean notify) {

   DrawingSupplier supplier = getDrawingSupplier();
     if (supplier != null) {
        stroke = supplier.getNextOutlineStroke();

   this.outlineStrokeList.setStroke(series, stroke);
         if (notify)
             fireChangeEvent();
     }
 }
}
```

```
class RendererImpl extends AbstractRenderer {
  public Paint lookupSeriesPaint(int series) {
     Paint seriesPaint = this.paintList.getPaint(
   series);
     if (seriesPaint == null && this.
   autoPopulateSeriesPaint) {
         DrawingSupplier supplier =
   getDrawingSupplier();
         if (supplier != null) {
             seriesPaint = supplier.getNextPaint();
             this.paintList.setPaint(series,
   seriesPaint);
          }
          if(this.basePaint == null)

   throw new Exception("Null 'basePaint'.");
          else
              refresh(this.basePaint);
      }
     if (seriesPaint == null) {
        seriesPaint = this.basePaint;
     }
   return seriesPaint;
 }

  public Paint lookupSeriesFillPaint(int series) {
     Paint seriesFillPaint = this.fillPaintList.
   getPaint(series);
     if (seriesFillPaint == null && this.
   autoPopulateSeriesFillPaint) {
         DrawingSupplier supplier =
   getDrawingSupplier();
         if (supplier != null) {
             seriesFillPaint = supplier.
   getNextFillPaint();
             this.fillPaintList.setPaint(series,
   seriesFillPaint);
         }
         if(this.baseFillPaint == null)

    throw new Exception("Null 'baseFillPaint'.");
         else
             refresh(this.baseFillPaint);
     }
      if (seriesFillPaint == null) {
          seriesFillPaint = this.baseFillPaint;
      }
      return seriesFillPaint;
  }

  public Stroke lookupSeriesOutlineStroke(int
   series) {
     Stroke result = this.outlineStrokeList.
   getStroke(series);
      if (result == null && this.
   autoPopulateSeriesOutlineStroke) {
         DrawingSupplier supplier =
   getDrawingSupplier();
         if (supplier != null) {
             result = supplier.getNextOutlineStroke
   ();
             this.outlineStrokeList.setStroke(
   series, result);
         }
         if(this.baseOutlineStroke == null)

   throw new Exception("Null 'baseOutlineStroke'.");
         else
            refresh(this.baseOutlineStroke);
     }
     if (result == null) {
        result = this.baseOutlineStroke;
      }
     return result;
  }
}
```

Figure 4.2: An intermediate version ChgCutter generates by applying the bug-fix (left) and refactoring (right) edits separated from a composite change in Figure 4.1. The highlighted portions are edited by ChgCutter.

4.2 ChgCutter: Decomposing Composite Changes for Code Review and Regression Testing

We present a change decomposition approach to help developers inspect a composite change and run tests on a decomposed, related change set.

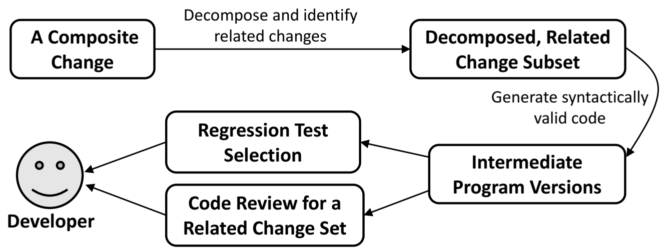

A Composite Change
Decompose and identify related changes
Decomposed, Related Change Subset
Generate syntactically valid code
Intermediate Program Versions
Regression Test Selection
Code Review for a Related Change Set
Developer

Figure 4.3: Overview of ChgCutter's workflow.

ChgCutter applies a change set to an original version of the program and generates an intermediate version, while applying a test selection technique to run an affected subset of the test suite on a decomposed change. Figure 4.3 depicts our approach consisting of the following three phases. (1) To help developers effectively inspect a composite change, ChgCutter allows developers to specify code change fragments within a composite change. By analyzing the portions of the user-selected changes that are often syntactically incomplete, it automatically identifies related change subsets separated from non-related changes. (2) To enable developers to investigate changes of interest, ChgCutter automatically generates intermediate program versions edited from an original program with related change subsets. To validate decomposed change subsets during regression testing, it creates syntactically correct versions of a program, collecting and ordering the prerequisites of changes. (3) To provide confidence that the change subset edited in an intermediate version behaves as intended and that the unchanged parts are not adversely affected by the change, we integrate with a regression test selection technique, automatically selecting a subset of the test suite affected by a change subset.

4.2.1 Decomposing Composite Changes

Decomposition by Using Program Slicing. To decompose a composite change, ChgCutter allows a developer to select a change region within a large diff output which involves composite changes. Given the user-selected change portion, ChgCutter parses code fragments into AST representation, and relates change regions within a same method by using static analysis. An interprocedural slicing technique is utilized to extract dependent program elements including all preceding dependent AST nodes based on transitive dependencies [48]. Based on selected change regions, the data and control dependency analysis [84] is performed to extract dependent codes—*dependent context* in the control flow graph. By using the backward static program slicing, the analysis results in a subset of statements (i.e., slice) in dependent context, which affects to value of variables in change regions. Thus, the dependent context enables ChgCutter to cluster change regions if two regions are related with data and control dependent relationship. ChgCutter identifies statically dependent regions that are likely to implement the same feature. This assumption has been used in several studies [17, 27, 38, 64] that applied program slicing to detect consistent concerns. Once change regions are grouped in the same method together, the grouped changes becomes an input information to search for other changes that are related code fragments distributed in different methods. We describe below a technique that obtains related parts further by generalizing a set of dependent code regions.

Decomposition by Using Program Element Generalization. Given an initial user-selected change region and dependent context, ChgCutter creates an initial edit *matching* template. Similar to the previous approach [108], a template can be generalized by replacing program elements such as identifiers (e.g., type, variable and method) and statements with parameters. The template generalization can increase a size of change subsets during ChgCutter's decomposition. Based on an edit matching template, a code reviewer can generalize identifiers into parameters that can be equivalent to different identifiers during our matching analysis. Recall the example from Figure 4.1, the variable name `seriesPaint` and method name `getSeriesPaint` are, for instance, generalized by a reviewer. These identifiers are mapped to each parameter `$VAR1` and `$VAR2`, whose all references are automatically parameterized by ChgCutter. As a result, ChgCutter parameterizes the AST node `Paint seriesPaint = getSeriesPaint(series)` into the node `Paint $VAR1 = $VAR2(series)`, which can be matched with the AST node `Paint seriesFillPaint = getSeriesFillPaint(series)` in the method `lookupSeriesFillPaint`. Therefore, these changes related to the refactorings can be separated from other changes, such as bug-fixes, in other locations of methods `lookupSerie- sFillPaint` and `lookupSeriesOutlineStroke`.

Decomposition By Using Tree-based AST Search. ChgCutter's code search technique is based on an AST matching technique using the RTED tree edit distance algorithm [74]. As previous studies [10, 11, 96], we parse a program to produce AST representation of the source program. ChgCutter takes as input an *input-subtree* and identifies a set of *output-subtrees* which is involved in exact or close matches of subtrees by comparing with an input-subtree in the generated AST trees. An input-subtree is produced by parsing an edit matching template. As a template consists of pre- and post-edit information, a pair of two input-subtrees is used to search for similar subtrees (i.e., output-subtrees) from the original and edited versions of a program. For similarity detection on AST trees, computing tree edit distances is typically used as an approximation algorithm [28, 107]. However, efficient tree similarity detection still remains an open problem, since our generalization technique for program elements needs to detect similarity between tokens of a pair of AST nodes.

To address this similarity detection problem, we combine a tree matching algorithm [74] with the word-mode diff function[3] that supports a differencing algorithm [71]. Continuing with our motivating example, an AST node `getSeriesPaint(series)` in an input-subtree is aligned with an AST node `getSeriesFillPaint(series)` in an output-subtree by computing the tree edit distance with a tree matching algorithm. We further align and compute similarity between a pair of tokens such as {("`getSeriesPaint`", "`getSeriesFillPaint`"), ("`(`", "`(`"), ("`series`", "`series`"), ("`)`", "`)`")}. We find there exists one nonequivalent token pair in the aligned token pairs; however, if the token "`getSeriesPaint`" in the AST node is replaced with a parameter `$VAR2`, we consider the token pair as equivalent.

Based on searched AST subtrees by applying an efficient tree matching algorithm and the AST node parameterization, ChgCutter finds related changes and decomposes them into related change subsets in other locations in the codebase. It differs from existing program differencing and search techniques that can find only concrete differences without much abstraction.

[3]`https://code.google.com/archive/p/google-diff-match-patch/`

4.2.2 Constructing Intermediate Versions

ChgCutter utilizes Eclipse JDT[4] built as a software component in the Eclipse framework. By using JDT APIs, ChgCutter constructs an intermediate version guaranteed to be syntactically correct. We make use of the AST rewrite infrastructure in Eclipse JDT for applying required edits to the AST nodes of a program.

Extracting AST Differences. To compute differences of original and edited versions of a program, ChgCutter leverages an AST differencing tool, called ChangeDistiller [36]. ChangeDistiller computes AST tree differences between pre- and post-edit source versions. We use ChangeDistiller, since it extracts fine-grained source changes at the level of statements (e.g., method invocation or variable assignment statements), comparing the AST representations between a pair of different versions of a program. The extracted differences then are represented as tree edit operations that are required to transform the original version of the program to the intermediate version. In Figure 4.4, for example, we illustrate the required edit operations for generating intermediate versions: four deletion operations from the original version, and four insertion operations into the edited version.

[4]http://eclipse.org/jdt/

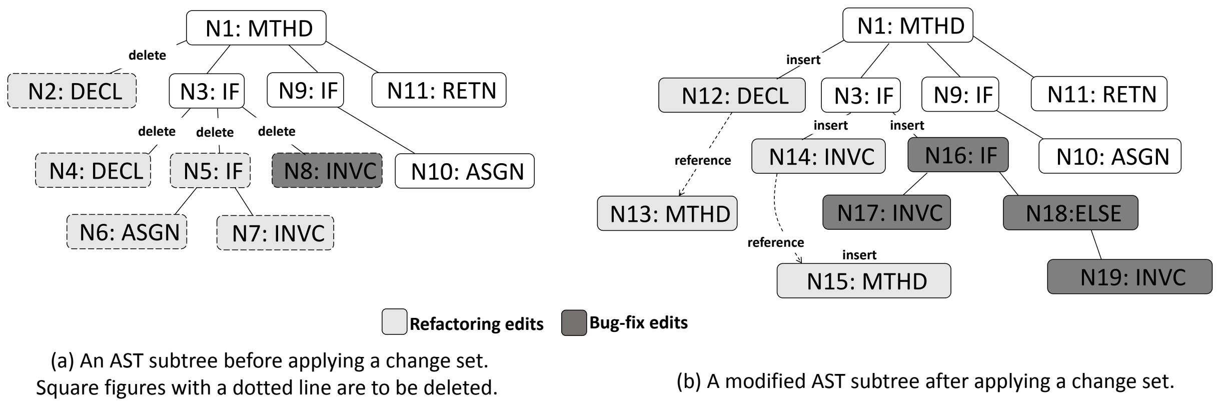


Figure 4.4: Applying required edit operations to generate an intermediate version for Figure 4.2 by using either refactoring edits or bug-fixing edits separated from a composite change in Figure 4.1.

Applying Edit Operations. ChgCutter reuses a tree matching algorithm to find edit locations by computing maximum common subtrees between an edit matching template and the region to be edited. As a template is partially generalized by a code reviewer, ChgCutter matches the abstract context against the location to be updated and reconstructs program elements for concrete change instances.

Continuing with our example, ChgCutter concretizes the variables within the parameterized statement `$VAR1 = this.$VAR2$.$VAR3(series)`, and generates the corresponding statement `seriesPaint = this.paintList.getPaint(series)` in method `lookupSeriesPaint` in the original version. To find concrete program elements (e.g., variable, method, and type), ChgCutter traverses AST trees, collects reference bindings by using the JDT framework, and constructs a hash table for matches between abstract parameters and concrete identifiers. It also produces the related, concrete statements, such as, `seriesFillPaint = this.fillPaintList.getPaint(series)` and `result = this.strokeList.getStroke(series)` in the methods `lookupSeriesFillPaint` and `lookupSeriesStroke`, respectively.

Similarly, ChgCutter maintains another AST hash table in the edited version to concretize variables in the parameterized statement `$VAR1 = $VAR2(series)`, and produces `seriesPaint = getSeriesPaint(series)` in method `lookupSeriesPaint`. It also finds and produces the related, concrete statements, such as, `seriesFillPaint = getSeriesFillPaint (series)` and `result = getSeriesStroke(series)` in methods `lookupSeriesFillPaint` and `lookupSeriesStroke`, respectively.

To apply an insertion operation to the suitable location of the original version, ChgCutter needs to find common ancestor AST node based on the sibling AST nodes that ChgCutter has identified. To find the lowest common ancestor AST node, ChgCutter compares the corresponding sibling AST nodes and computes the distance between the root node (i.e., method head) and each sibling node. To apply an deletion operation, ChgCutter deletes the AST node from the original version only if there exists no referenced statement. To apply an replacement operation, ChgCutter finds an AST node in the original version and replaces it by referring to an edit matching template. ChgCutter automatically rewrites a pre-edit version's AST, which leads to the corresponding intermediate version. It then unparses the resulting ASTs into source code, which is recompiled by Eclipse JDT. The recompiled program runs with a test suite to determine whether a related change subset causes a program to produce incorrect results. We will describe details below.

4.2.3 Validating Intermediate Versions

ChgCutter generates an intermediate version by applying an identified subset of related changes to the original version of the program. Regression testing is then applied to the intermediate version to provide confidence that the change subset behaves intended and that the unchanged parts of the program have not been adversely influenced by the change subset.

ChgCutter leverages a test selection technique [68] to identify a subset of the test suite affected by the change subset applied to the intermediate version. Unlike previous test selection approaches [43, 79], our approach does not analyze a control-flow-based representation of both original and edited versions of the program to select the test cases to be rerun.

We construct a call graph for each test in the test suite that was used to test the original version of the program P. We obtain dynamic call graphs by tracing the execution of the tests. For a given set T of the regression test suite, our approach determines a subset T' of the entire tests that is potentially affected by the change subsets—related atomic changes A that ChgCutter identified above. We correlate the change subsets against the dynamic call graphs for the tests in T in the original version of the program to select a subset T'.

The call graphs we have constructed contain one node for each method, and edges between nodes to represent call references between methods. For example, our approach constructs the call graphs before the changes have been applied to an intermediate version that we have created. We determine an affected test, if its call graph in the original version of the program either includes a node that corresponds to a changed method (CM) or deleted method change (DM). We also determine an affected test by checking a changed call by an overridden method or a hierarchy change, such as lookup change (LC). We define formally the equations to describe how we find affected tests T'.

$$\begin{aligned} AffectedTests(T,A) = \;& \{t_i | t_i \in T, Nodes(P,t_i) \cap (CM \cup DM) \neq \emptyset\} \cup \\ & \{t_i | t_i \in T, m \in Nodes(P,t_i), A.m \rightarrow B.m/C.m \in Edges(P,t_i)\} \end{aligned} \tag{4.1}$$

CM represents any change to a method's body. A call reference is defined as $A.m \rightarrow B.m/C.m$, indicating possible control flow from method $A.m$ to method $B.m$ due to a method $C.m$ on an object type C. To obtain the call graphs, we utilized an aspect-oriented programming (AOP) technique [57, 58] for instrumenting the class files of the original version of the program and their tests. For the reuse of analysis results, call graphs are stored as XML files. Executing each test case that has been instrumented by the AOP tool produces an XML file containing the program's dynamic call graph. Our approach, thus, helps developers selectively executes test cases to quickly detect faults in the edited version of the program.

4.3 Evaluation

The evaluation of our approach aims to answer the following research questions. To answer these questions, we conduct an exploratory study to understand how effective ChgCutter is during code review and regression testing.

- RQ3: Can ChgCutter accurately *construct* syntactically valid intermediate versions by decomposing a composite change?

- RQ4: Can ChgCutter accurately *select* a subset of the regression tests to validate each intermediate version?

The purpose of RQ3 is to determine whether the static analysis result computed by ChgCutter is capable of decomposing a composite change and grouping a subset of related changes with dependent program elements as well as similar code changes, particularly in cases where an intermediate version are required to be compiled and runnable without developer intervention. The rational behind RQ4 is to determine whether our test selection approach is applicable with the code review technique to validate intermediate versions. We are interested in the number of test cases selected to test an intermediate version, which are reasonably lower compared to all test cases for the post-edit version. This also aims to determine whether selection accuracy are acceptable. The selection accuracy will be estimated by precision, the percentage of correctly identified test cases compared to all found tests, and by recall, the percentage of correct test cases out of all expected tests.

4.3.1 Experimental Design

To evaluate our approach, we apply ChgCutter to four open source projects, including JFreeChart—a graph and chart library implemented in Java; Apache Tomcat—a Java implementation for Servlet, JavaServer Pages and WebSocket technologies; ArgoUML—an UML diagramming tool, and; Eclipse JDT—Eclipse Java development tool.[5]

We developed a mining strategy to obtain the ground truth data set from the four subject applications. We manually classified individual change sets and checked whether they address multiple development issues (composite changes). To mine change history, we

[5]www.jfree.org, tomcat.apache.org, argouml.tigris.org, and eclipse.org/jdt

wrote a batch script which runs a diff utility to compare original and edited versions of the program and produces diff patch files containing changes such as addition, deletion, and modification. Our script program then divides each diff file into a set of change hunks.[6] Lastly, for the investigation of recurring similar changes in the data set, we use CCFinder [54] a clone detection tool that alleviates our manual work—change set classification. Based on the output of the clone detector, we manually decompose and group the resulting clones as a set of related changes. Also, commit log messages mentioning more than one issues are examined during our investigation.

As a test selection ground truth, regression test suites were generated for our dataset. The use of randomly generated test suites is not always used in real projects. Still, random testing has recently been a cost-effective alternative due to the available tool support. We used an automated test generator Randoop[7] for two reasons. First, Randoop is a state-of-art automatic unit test generator for Java applications using feedback-directed random generation. Second, it has been widely used for validating diverse changes, including refactorings [85, 86], in open source projects. We used the same Randoop configuration in all test suite generations (time limit is 2 seconds, and maximum test size is 500 statements). We totally obtained 3,456 tests for original versions used for each intermediate version in the dataset. The experiment was conducted on a machine with a quad-core 2.2GHz CPU and 16GB RAM.

[6]A hunk is a single modification unit of the region regarding two versions.
[7]randoop.github.io

Table 4.1: The default ChgCutter's results before generalization. %P1 and %R1 shows the precision and recall for the intermediate version generation, indicating the percentage of correctly identified change locations compared to all found location and the percentage of correct change locations out of all expected locations, respectively. %P2 and %R2 show the precision and recall for the test case selection, indicating the percentage of correctly identified test cases compared to all found test cases and the percentage of correct test cases out of all expected test cases, respectively.

	Ground Truth Data Set				**Intermediate Version Generation**							**Regression Test Selection**					
Index	GT1	GT2	GT3	GT4	RS1	RS2	RS3	%P1	%R1	%F1	%Sim	RS4	RS5	%P2	%R2	%F2	RT
1	16	2	130	22	10	10	2	100	62.5	76.9	92.8	13	13	100	59.1	74.3	1,241
2	27	4	65	11	5	5	4	100	18.5	31.3	65.3	2	2	100	18.2	30.8	638
3	44	3	117	38	4	4	3	100	9.1	16.7	34.7	3	3	100	7.9	14.6	1,747
4	6	2	234	31	2	2	2	100	33.3	50.0	45.1	16	16	100	51.6	68.1	1,153
5	16	3	333	71	3	3	3	100	18.8	31.6	59.7	11	11	100	15.5	26.8	819
6	15	3	355	317	5	5	3	100	33.3	50.0	68.7	120	120	100	37.9	54.9	386
7	16	4	650	127	8	8	4	100	50.0	66.7	80.7	30	30	100	23.6	38.2	2,057
8	15	2	174	10	2	2	2	100	13.3	23.5	52.2	2	2	100	20.0	33.3	312
9	12	1	103	58	1	1	1	100	8.3	15.4	83.3	10	10	100	17.2	29.4	602
10	4	1	407	17	1	1	1	100	25.0	40.0	56.4	8	8	100	47.1	64.0	1,932
11	19	3	888	112	3	3	3	100	15.8	27.3	57.1	15	15	100	13.4	23.6	3,455
Total	190	28	3,456	814	44	44	28	100	23.2	37.6	63.3	230	230	100	28.3	44.1	14,342

Table 4.2: The ChgCutter$_{gen}$'s results with generalization. TYPE denotes a type of identifier parameterizations: V (variable), M (method name), T (type), and E (statement exclusion).

	Ground Truth Data Set					**Intermediate Version Generation**							**Regression Test Selection**					
Index	GT1	GT2	GT3	GT4	TYPE	RS1	RS2	RS3	%P1	%R1	%F1	%Sim	RS4	RS5	%P2	%R2	%F2	RT
1	16	2	130	22	V,M,T,E	18	16	2	88.9	100	94.1	100	48	22	45.8	100	62.8	1,206
2	27	4	65	11	V,E	29	27	4	93.1	100	96.4	100	37	11	29.7	100	45.8	603
3	44	3	117	38	V	44	44	3	100	100	100	100	38	38	100	100	100	1,165
4	6	2	234	31	V	5	5	2	100	83.3	90.9	93.1	30	30	100	96.8	98.4	1,131
5	16	3	333	71	V	11	11	3	100	68.8	81.5	99.8	45	45	100	63.4	77.6	793
6	15	3	355	317	V, E	13	13	3	100	86.7	92.9	99.8	312	312	100	98.4	99.2	163
7	16	4	650	127	V	13	13	4	100	81.3	89.7	99.7	101	101	100	79.5	88.6	1,195
8	15	2	174	10	V	15	15	2	100	100	100	100	10	10	100	100	100	160
9	12	1	103	58	V,M,T,E	12	12	1	100	100	100	100	58	58	100	100	100	437
10	4	1	407	17	V,M	4	4	1	100	100	100	100	17	17	100	100	100	1,906
11	19	3	888	112	V,E	18	18	3	100	94.7	97.3	99.8	93	93	100	83.0	90.7	3,448
Total	190	28	3,456	814		182	178	28	97.8	93.7	95.7	99.3	789	737	87.6	90.5	89.0	12,207

4.3.2 Study Results and Discussion

We use two variants of the change investigation for our evaluation. The default ChgCutter searches the related change set without parameterization. ChgCutter$_{gen}$ parameterizes variables, types and method names, or excludes statements to iteratively search the related change set. We apply the default ChgCutter and ChgCutter$_{gen}$ to the data sets. Tables 4.1 and 4.2 present the results. Regarding validation process, I investigated ChgCutter's results. The remaining authors then analyzed the results in the meetings. When there was lack of consensus, the issues were put to the next analysis round, and a mutual decision was made.

In Tables 4.1 and 4.2, each task has a unique data set ID. GT1 represents the number of the change instances that a developer commits to a VCS repository, and GT2 the number of the change sets. For example, the data set #1 contains a composite change which addresses two independent issues (see GT2), including 16 code blocks (i.e., bodies of statements **`if`**, **`for`**, etc.) that were changed (see GT1). GT3 denotes the number of the total tests Randoop generates, and GT4 the number of the expected tests to be selected.

RS1 means the number of the change instances that ChgCutter identifies; RS2 the number of the change instances that ChgCutter correctly identifies; RS3 the number of the intermediate versions that ChgCutter generates with no syntactic violation. %P1 and %R1 shows the precision and recall of the ChgCutter's capability for the intermediate version generation, indicating the percentage of correctly identified change locations compared to all found location and the percentage of correct change locations out of all expected locations, respectively. %F1 calculates the accuracy by using the harmonic mean of %P1 and %R1. The syntactic similarity %Sim in the next column is measured by comparing the intermediate version that ChgCutter generates and the version that a real developer has implemented.[8]

RS4 in the sixth column from the last denotes the number of test cases that ChgCutter identifies, and RS5 in the next column shows the number of test cases that ChgCutter correctly finds. %P2 and %R2 show the precision and recall of the ChgCutter's capability for the test case selection, indicating the percentage of correctly identified test cases compared to all found test cases and the percentage of correct test cases out of all expected test cases, respectively. %F2 calculates the test selection accuracy by the harmonic mean of %P2 and %R2.

RQ3. The ChgCutter's capability for the intermediate version generation? We assess ChgCutter's precision by examining how many of decomposed change sets of the intermediate versions are indeed true decomposed intermediate versions. ChgCutter$_{gen}$ builds 28 intermediate versions by the change decomposition, 26 of which are correct, resulting in 97.8% precision. Regarding recall, ChgCutter$_{gen}$ builds 93.7% of all ground truth data sets. It generates intermediate versions, separating a composite change with 95.7% accuracy and 99.3% similarity.

Our approach generates intermediate versions that are not easy to build because they require running on the test suite without both compilation and runtime errors. For example, our approach combines the prerequisites of the individual change subsets, which are processed as a single set. Based on prerequisites in a dependence chain, it ensures the AST

[8]The *Levenshtein* edit distance [69] is used to measure the similarity between two sequences of characters based on number of deletions, insertions, or substitutions required to transform one sequence to the other.

construction of syntactically correct intermediate versions of a program.

False Positives. No false positive in the intermediate versions are reported by the default ChgCutter, since the default ChgCutter builds the intermediate versions by finding only edit locations with exactly the same context with the AST edit matching template. Four intermediate versions are incorrectly generated by ChgCutter$_{gen}$ due to semantically non-related changes. For example, Extract Method [37] is applied to a clone group in Apache Tomcat (r980410). Although ChgCutter$_{gen}$ identifies the related changes, it also finds other non-related changes which have similar AST structure. Semantic similarity analysis examining program behavior can prevent this false positive, which will be combined in our heuristics in the future.

False Negative. The intermediate versions generated by the default ChgCutter do not often comprise related changes, because the default edit matching template cannot identify ASTs with different identifiers despite similar structures. Most intermediate versions generated by ChgCutter$_{gen}$ contain related changes due to the matching technique with parameterization. However, 16 related changes are not identified in JFreeChart (r1424), since the tree matching algorithm produces misalignment between AST nodes. This limitation can be overcome by plugging in tree edit distance algorithms that are more resilient to differences in structure.

RQ4. The ChgCutter's capability for the regression test selection? We estimate the precision of ChgCutter by evaluating how many of the identified test cases are indeed a true affected test cases. As we determine the effects of source code modification and identify a related subset of the test suite, we consider any test case as an affected test case, if its call graph correlates the decomposed changes in an intermediate version. ChgCutter$_{gen}$ identifies 789 affected tests, 737 of which are correct, resulting in 87.6% precision. Regarding recall, ChgCutter$_{gen}$ identifies 90.5% of all ground truth data sets. It identifies 789 out of 3,456 affected tests with 89.0% accuracy, and reduces both the time by over 12 seconds and the number of tests by over 78% that are required to perform regression testing in total on the generated intermediate versions.

Our approach helps developers not only understand a composite change but also conduct debugging on its modified behavior after a test may fail unexpectedly. It automatically classifies tests whether each test should be run for related atomic changes separated from a composite change, which is not easy to determine since performing regression testing requires revealing the faults by this affected subset safely—the same as those revealed by running the entire test suite.

False Positives. No false positive is found in test cases identified by the default ChgCutter; however, the result reports affected test cases only 230 out of 814. Most tests are incorrectly identified by ChgCutter$_{gen}$ due to the lack of capability to classify semantic differences between atomic changes in an intermediate version. As we discuss above, four intermediate versions are a false positive. In these versions, our approach identifies 26 tests out of 995 that Randoop generates for classes `CoyoteAdapter` and `Response` in Apache Tomcat (r980410). These test cases identified by non-related changes have influenced the low precision with respect to the data set 1 and 2. Comparison of control flow graphs [79] can prevent this false positive, which is our future work.

False Negatives. The subset of regression tests identified by the default ChgCutter often miss affected tests, because the affecting changes in intermediate versions are related to additions and deletions without abstraction levels, while searching for change instances based on concrete constraints. In other words, it does not capture test cases from the test suite that reveal a fault in the program version with a composite change that may use different identifier names. Most tests identified by ChgCutter$_{gen}$ exercise decomposed changes applied to intermediate versions. However, for the data sets 5, 7 and 11, ChgCutter$_{gen}$ may be too conservative in the test selection compared to others, because there are few identified related changes in classes `ClusteringSingleSignOn` and `ChartFactory` in Apache Tomcat (r980410) and JFreeChart (r1424), respectively due to the misalignment for tree pairs with different AST subtree shapes during the tree edit distance computation. A specific pair of subtrees is to be resolved in our heuristics in the future.

4.3.3 Threats to Validity

Our evaluation with four open projects may not be generalizable to other projects. As the limitation of number and coverage of the existing tests, we make use of an automated test generation tool; however, it is a cost-effective alternative to generate test cases for the methods impacted by changes. In our empirical study, we measured the accuracy to identify related atomic change sets and the code completeness (i.e., similarity to expected changes) to build a syntactically valid intermediate version. Other measures such as the degree of the tool effectiveness and satisfaction could be used to measure developer usability and productivity for code review and regression testing. In our case study, we used large and small code changes with diverse change types, including bug-fixes and refactorings to mitigate potential subjectivity bias. Since the goal of ChgCutter is to help understand a composite change and test individual, cohesive change set, the changes usually pertain to the modifications about similar and dependent changes, instead of general program modifications.

CHAPTER 5

CONCLUSIONS and FUTURE RESEARCH

5.1 Conclusions

In this research, we develop a synergistic approach by combining static and dynamic analysis techniques to address fundamental problems in software quality assurance, particularly enabling testing techniques to offer a code review tool an attractive complement such as accurate execution monitoring results. Our approach helps developers sufficiently understand software systems after these systems are modified during quality assurance tasks. Understanding a system's behavior implies studying such artifacts as source code and its changes, which is tedious and error-prone. Also, one of the most expensive activities is the testing as software is developed and maintained. To improve programmers productivities and reduce development and maintenance costs, we combine static and dynamic analyses to facilitate both activities, testing and code reviews, by making easier to translate approaches from one activity to the other. Rather than using either purely static or purely dynamic analysis, we synergize both the soundness of static analysis and the accuracy of dynamic analysis to obtain a new, hybrid analysis technique for (i) confidence that the changes behave as intended and for (ii) a sufficient level of comprehension of a system's inner behavior during a given maintenance task.

First, we present a novel formal approach to select tests for regression testing on EFSMs. Our approach selects tests that are guaranteed to exercise changes to EFSMs while discarding those that are guaranteed not to exercise the changes, without actually executing the tests. It identifies a class of fully-observable tests whose descriptions contain all the information about the EFSM transitions executed by the test, formulating a structural invariant to characterize these tests. In our evaluation, we show that our approach achieves better efficiency in comparison to brute-force symbolic execution approaches as well as dependency based approaches. We also present an approach, called ChgCutter, to decomposing a composite change and identifying a subset of related atomic changes. By applying edits of the identified change subset to an original version, our approach automatically generates an intermediate version which can be tested during regression test. Given a generated intermediate version, our approach selects and runs an affected subset of the test suite, and reduces the time to perform regression testing. To assist developers during development, our approach has been integrated closely with Eclipse (www.eclipse.org), a widely used open-source development environment. In our evaluation, we assess our technique with four open source projects and find that it helps developers investigate a composite code change for both program understanding and debugging.

Effective maintenance and development activities require a significant amount of efforts from the developers to thoroughly comprehend and validate code changes. Our approach demonstrates that the large amount of the efforts devoted to the assurance activities can be improved with these hybrid analysis approaches, complementing the major challenges and enhancing one another by providing information that would otherwise be unavailable.

Our research has great potential to fundamentally impact the software quality improvement during peer code reviews and testings by effectively managing the software evolution.

5.2 Future Research

User study to improve tool usability. Plans for future research include user study with professional developers to improve our tool's usability. We plan to conduct interviews with participants from PayPal.com to understand the current challenges they face during the code change inspection; and to study whether and how ChgCutter could help participants. Because we are interested in whether participants accept decomposed intermediate versions, the participants are asked to partition composite changes. If an intermediate version result of our approach matches one of the ways humans decompose the composite change, then this result can be considered acceptable. First we will give a presentation to introduce ChgCutter's features. This presentation includes demo of how to use ChgCutter Eclipse plug-in. For individual participates, we will conduct a semi-structured interview to collect their feedback on the utility of ChgCutter. We will audio-record the interview and analyze the feedback. The interview questions may be described as below.

1. How often do you have composite changes during code review?
2. What kind of challenges do you face composite changes during code review?
3. In which situation, do you think ChgCutter can effectively and efficiently improve code review process?
4. How do you like or dislike ChgCutter?

Code visualization for intermediate version to help undo/redo selected changes. During code review, one change set may not be sufficient to understand all changes. To better understand changes, a table view presents all change sets in a list of rows that are divided into sections. Each section has edit operations for one change set. User can select one or more change sets to build an intermediate version by applying their edit operations. Moreover, a change set can be removed from an intermediate version by undoing. We will extend CHGCUTTER Eclipse plug-in to handle code visualization for intermediate version to leverage user undo/redo selected changes.

References

[1] Yices (an smt solver). http://yices.csl.sri.com/.

[2] Z3 (an thoerem prover). http://research.microsoft.com/en-us/um/redmond/projects/z3/.

[3] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski. Software inspections: An effective verification process. *IEEE Software*, 6(3):31–36, 1989.

[4] R. Adams, W. Tichy, and A. Weinert. The cost of selective recompilation and environment processing. *ACM Transactions Software Engineering Methodol.*, 3(1):3–28, 1994.

[5] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, 1998.

[6] A. Bacchelli and C. Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 International Conference on Software Engineering*, pages 712–721. IEEE Press, 2013.

[7] B. S. Baker. A program for identifying duplicated code. 1993.

[8] B. S. Baker. On finding duplication and near-duplication in large software systems. In *Proceedings of 2nd Working Conference on Reverse Engineering, 1995.*, pages 86–95, 1995.

[9] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, volume 1, pages 134–144. IEEE, 2015.

[10] I. D. Baxter, C. Pidgeon, and M. Mehlich. Dms®: Program transformations for practical scalable software evolution. In *Proceedings of the 26th International Conference on Software Engineering*, pages 625–634. IEEE Computer Society, 2004.

[11] I. D. Baxter, A. Yahin, L. Moura, M. S. Anna, and L. Bier. Clone detection using abstract syntax trees. In *Proceedings of International Conference on Software Maintenance, 1998.*, pages 368–377. IEEE, 1998.

[12] H. C. Benestad, B. Anda, and E. Arisholm. Understanding software maintenance and evolution by analyzing individual changes: a literature review. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(6):349–378, 2009.

[13] A. Bertolino. Software testing research: Achievements, challenges, dreams. In *2007 Future of Software Engineering*, pages 85–103. IEEE Computer Society, 2007.

[14] L. C. Briand, Y. Labiche, and S. He. Automating regression test selection based on uml designs. *Information and Software Technology*, 51(1), 2009.

[15] L. C. Briand, Y. Labiche, and G. Soccar. Automating impact analysis and regression test selection based on uml designs. In *International Conference on Software Maintenance*, 2002.

[16] E. Bringmann and A. Kramer. Model-based testing of automotive systems. In *1st International Conference on Software Testing, Verification and Validation (ICST'08)*, 2008.

[17] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca. Decomposing legacy programs: A first step towards migrating to client–server platforms. *Journal of Systems and Software*, 54(2):99–110, 2000.

[18] R. Y. Chang, A. Podgurski, and J. Yang. Discovering neglected conditions in software by mining dependence graphs. *IEEE Transactions on Software Engineering*, 34(5):579–596, 2008.

[19] Y. Chen, R. L. Probert, and D. P. Sims. Specification-based regression test selection with risk analysis. In *Conference of the Centre for Advanced Studies on Collaborative research (CASCON'02).* IBM Press, 2002.

[20] Y. Chen, R. L. Rrobert, and H. Ural. Regression test suite reduction using extended dependence analysis. In *4th International Workshop on Software Quality Assurance (SOQUA'07)*, 2007.

[21] O. C. Chesley, X. Ren, and B. G. Ryder. Crisp: a debugging tool for java programs. In *Proceedings of the 21st IEEE International Conference on Software Maintenance, 2005 (ICSM'05).*, pages 401–410, 2005.

[22] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[23] K. D. Cooper, K. Kennedy, and L. Torczon. Interprocedural optimization: Eliminating unnecessary recompilation. In *Proceedings of the 1986 SIGPLAN Symposium on Compiler Construction*, SIGPLAN '86, pages 58–67. ACM, 1986.

[24] B. Cornelissen, A. Zaidman, A. van Deursen, L. Moonen, and R. Koschke. A systematic survey of program comprehension through dynamic analysis. *Software Engineering, IEEE Transactions on*, 35(5):684–702, 2009.

[25] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. An efficient method of computing static single assignment form. In *16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL'89)*, 1989.

[26] B. Daniel and Z. Pitro. On communicating finite-state machines. *Journal of the ACM*, 30(2), 1983.

[27] A. De Lucia, A. R. Fasolino, and M. Munro. Understanding function behaviors through program slicing. In *Proceedings of Fourth Workshop on Program Comprehension, 1996.*, pages 9–18. IEEE, 1996.

[28] E. D. Demaine, S. Mozes, B. Rossman, and O. Weimann. An optimal decomposition algorithm for tree edit distance. *ACM Transactions on Algorithms (TALG)*, 6(1):2, 2009.

[29] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *Journal of the ACM*, 52(3), 2005.

[30] M. Dmitriev. Language-specific make technology for the java programming language. In *Proceedings of the 17th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '02, pages 373–385. ACM, 2002.

[31] H. Do, S. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering*, 10(4), 2005.

[32] A. Dunsmore, M. Roper, and M. Wood. Practical code inspection techniques for object-oriented systems: An experimental comparison. *IEEE Software*, 20(4):21–29, 2003.

[33] M. E. Eagan. Advances in software inspections. *IEEE Transactions Software Engineering*, 12(7):744–751, 1986.

[34] M. E. Fagan. Design and code inspections to reduce errors in program development. *IBM Syst. J.*, 38(2-3):258–287, 1999.

[35] R. Falke, P. Frenzel, and R. Koschke. Empirical evaluation of clone detection using syntax suffix trees. *Empirical Software Engineering*, 13(6):601–643, 2008.

[36] B. Fluri, M. Wursch, M. PInzger, and H. C. Gall. Change distilling: Tree differencing for fine-grained source code change extraction. *Software Engineering, IEEE Transactions on*, 33(11):725–743, 2007.

[37] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2000.

[38] K. B. Gallagher and J. R. Lyle. Using program slicing in software maintenance. *Software Engineering, IEEE Transactions on*, 17(8):751–761, 1991.

[39] T. L. Graves, M. J. Harrold, J. Kim, A. Porters, and G. Rothermel. An empirical study of regression test selection techniques. In *Proceedings of the International Conference on Software Engineering, 1998.*, pages 188–197, 1998.

[40] T. Guardian. Why we all sell code with bugs. Aug 2006.

[41] B. Guo, M. Subramaniam, and Z. Pap. In *20th International Conference on Testing of Software and Communication Systems and 8th International FATES Workshop (TestCom/FATES'08).*

[42] R. Gupta, M. J. Harrold, and M. L. Soffa. An approach to regression testing using slicing. In *Software Maintenance, 1992. Proceerdings., Conference on*, pages 299–308, 1992.

[43] M. J. Harrold, J. A. Jones, T. Li, D. Liang, A. Orso, M. Pennings, S. Sinha, S. A. Spoon, and A. Gujarathi. Regression test selection for java software. In *Proceedings of OOPSLA*, pages 312–326. ACM, 2001.

[44] M. J. Harrold and A. Orso. Retesting software during development and maintenance. In *Frontiers of Software Maintenance (FoSM'08)*, 2008.

[45] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with blast. In *10th International Conference on Model Checking Software*, 2003.

[46] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 121–130. IEEE Press, 2013.

[47] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proceedings of MSR*, 2013.

[48] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions Program. Language System*, 12(1):26–60, 1990.

[49] L. Jiang, G. Misherghi, Z. Su, and S. Glondu. Deckard: Scalable and accurate tree-based detection of code clones. In *Proceedings of the 29th International Conference on Software Engineering*, ICSE '07, pages 96–105. IEEE Computer Society, 2007.

[50] J. H. Johnson. Identifying redundancy in source code using fingerprints. In *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*, CASCON '93, pages 171–183. IBM Press, 1993.

[51] J. H. Johnson. Substring matching for clone detection and change tracking. In *Proceedings of International Conference on Software Maintenance, 1994.*, pages 120–126, 1994.

[52] J. H. Johnson. Visualizing textual redundancy in legacy source. In *Proceedings of the 1994 Conference of the Centre for Advanced Studies on Collaborative Research*, CASCON '94, pages 32–. IBM Press, 1994.

[53] N. Juristo, A. M. Moreno, and S. Vegas. Reviewing 25 years of testing technique experiments. *Empirical Software Engineering*, 9(1-2):7–44, 2004.

[54] T. Kamiya, S. Kusumoto, and K. Inoue. Ccfinder: A multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions Software Engineering*, 28(7):654–670, 2002.

[55] D. Kapur and H. Zhang. An overview of rewrite rule laboratory (rrl). In *3rd International Conference on Rewriting Techniques and Applications (RTA'89)*, 1989.

[56] C. Keum, S. Kang, I. Ko, J. Baik, and Y. Choi. Generating test cases for web services using extended finite state machine. In *18st International Conference on Testing of Software and Communication Systems (TestCom'06)*, 2006.

[57] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of aspectj. In *Proceedings of ECOOP*, pages 327–353. Springer-Verlag, 2001.

[58] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP*, pages 220–242. Springer-Verlag, 1997.

[59] J. C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7), 1976.

[60] B. Korel, G. Koutsogiannakis, and L. H. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *3rd International workshop on Advances in model-based testing*, 2007.

[61] B. Korel, G. Koutsogiannakis, and L. H. Tahat. Application of system models in regression test suite prioritization. In *IEEE International Conference on Software Maintenance (ICSM'08)*, 2008.

[62] B. Korel, L. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *18th IEEE International Conference on Software Maintenance (ICSM'02)*, 2002.

[63] B. Korel, L. H. Tahat, and M. Harman. Test prioritization using system models. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, 2005.

[64] F. Lanubile and G. Visaggio. Extracting reusable functions by flow graph based program slicing. *Software Engineering, IEEE Transactions on*, 23(4):246–259, 1997.

[65] D. Lee and M. Yiannakakis. Principles and methods of testing finite state machines - a survey. *Proceedings of the IEEE*, 84(8), 1996.

[66] M.-W. Lee, J.-W. Roh, S.-w. Hwang, and S. Kim. Instant code clone search. In *Proceedings of the Eighteenth ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '10, pages 167–176. ACM, 2010.

[67] M. Lejter, S. Meyers, and S. P. Reiss. Support for maintaining object-oriented programs. *IEEE Transactions Software Engineering*, 18(12):1045–1052, 1992.

[68] H. Leung and L. White. A cost model to compare regression test strategies. In *Proceedings Conference on Software Maintenance*, 1991.

[69] V. I. Levenstein. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady 10*, pages 707–710, 1966.

[70] Y. Lin, Z. Xing, Y. Xue, Y. Liu, X. Peng, J. Sun, and W. Zhao. Detecting differences across multiple instances of code clones. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 164–174. ACM, 2014.

[71] E. W. Myers. Ano (nd) difference algorithm and its variations. *Algorithmica*, 1(1-4):251–266, 1986.

[72] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of 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 383–392. ACM, 2009.

[73] A. K. Onoma, W.-T. Tsai, M. Poonawala, and H. Suganuma. Regression testing in an industrial environment. *Commun. ACM*, 41(5):81–86, 1998.

[74] M. Pawlik and N. Augsten. Rted: A robust algorithm for the tree edit distance. *Proceedings VLDB Endow.*, 5(4):334–345, 2011.

[75] G. Ramalingam and T. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1), 1996.

[76] P. Rigby, B. Cleary, F. Painchaud, M. A. Storey, and D. German. Contemporary peer review in action: Lessons from open source development. *IEEE Software*, 29(6):56–61, 2012.

[77] G. Rothermel and M. J. Harrold. Selecting tests and identifying test coverage requirements for modified software. In *ACM SIGSOFT International symposium on Software testing and analysis (ISSTA '94)*, 1994.

[78] G. Rothermel and M. J. Harrold. Analyzing regression test selection techniques. *IEEE Transactions on Software Engineering*, 22(8), 1996.

[79] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 6, 1997.

[80] C. K. Roy and J. R. Cordy. An empirical study of function clones in open source software. In *Reverse Engineering, 2008. WCRE '08. 15th Working Conference on*, pages 81–90, 2008.

[81] G. W. Russell. Experience with inspection in ultralarge-scale development. *IEEE Software*, 8(1):25–31, 1991.

[82] R. Santelices, P. K. Chittimalli, T. Apiwattanapong, A. Orso, and M. J. Harrold. Test-suite augmentation for evolving software. In *23rd IEEE/ACM International Conference on Automated Software Engineering (ASE'08)*, 2008.

[83] F. Shull and C. Seaman. Inspecting the history of inspections: An example of evidence-based technology diffusion. *IEEE Software*, 25(1):88–90, 2008.

[84] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 432–441. ACM, 1999.

[85] G. Soares. Making program refactoring safer. In *Proceedings of ICSE*, pages 521–522, 2010.

[86] G. Soares, R. Gheyi, and T. Massoni. Automated behavioral testing of refactoring engines. *IEEE Transactions on Software Engineering*, pages 147–162, 2013.

[87] M. Subramaniam and P. Chundi. An approach to preserve protocol consistency and executability across updates. In *6th International Conference on Formal Engineering Methods (ICFEM'02)*, 2004.

[88] M. Subramaniam and B. Guo. A rewrite-based approach for change impact analysis of communicating systems using a theorem prover. Technical report, Compute Science Department, Univerisity of Nebraska at Omaha, 2008.

[89] M. Subramaniam, B. Guo, and Z. Pap. Using change impact analysis to select tests for extended finite state machines. In *7th IEEE International Conference on Software Engineering and Formal Methods (SEFM'09)*, 2009.

[90] M. Subramaniam and Z. Pap. Analyzing the impact of protocol changes on tests. In *18th International Conference on Testing of Communicating Systems (TestCom'06)*, 2006.

[91] M. Subramaniam, L. Xiao, B. Guo, and Z. Pap. An approach for test selection for efsms using a theorem prover. In *21st International Conference on Testing of Software and Communication Systems and 9th International FATES Workshop (TestCom/FATES'09)*, 2009.

[92] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, FSE '12, pages 51:1–51:11. ACM, 2012.

[93] Y. Tao and S. Kim. Partitioning composite code changes to facilitate code review. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 180–190, 2015.

[94] W. F. Tichy. Smart recompilation. *ACM Transactions Program Language System*, 8(3):273–291, 1986.

[95] B. Vaysburg, L. H. Tahat, and B. Korel. Dependence analysis in reduction of requirement based test suites. In *ACM SIGSOFT International symposium on software testing and analysis (ISSTA '02)*, 2002.

[96] V. Wahler, D. Seipel, J. W. v. Gudenberg, and G. Fischer. Clone detection in source code by frequent itemset techniques. In *Proceedings of the Source Code Analysis and Manipulation, Fourth IEEE International Workshop*, SCAM '04, pages 128–135. IEEE Computer Society, 2004.

[97] S. Wang, D. Lo, and L. Jiang. Code search via topic-enriched dependence graph matching. In *Reverse Engineering (WCRE), 2011 18th Working Conference on*, pages 119–123, 2011.

[98] X. Wang, D. Lo, J. Cheng, L. Zhang, H. Mei, and J. X. Yu. Matching dependence-related queries in the system dependence graph. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, ASE '10, pages 457–466. ACM, 2010.

[99] S. Weileder. Parteg (partition test generator). http://parteg.sourceforge.net/.

[100] K. E. Wiegers. *Peer reviews in software: A practical guide*. Addison-Wesley Boston, 2002.

[101] N. Wilde, P. Matthews, and R. Huitt. Maintaining object-oriented software. *IEEE Software*, 10(1):75–80, 1993.

[102] W. E. Wong, J. R. Horgan, S. London, and H. Agrawal. A study of effective regression testing in practice. In *Proceedings of The Eighth International Symposium on Software Reliability Engineering, 1997.*, pages 264–274, 1997.

[103] Z. Xu and G. Rothermel. Directed test suite augmentation. In *Asia-Pacific Software Engineering Conference (APSEC'09)*, 2009.

[104] W. Yang and W. Yang. Identifying syntactic differences between two programs. *SOFTWARE - PRACTICE AND EXPERIENCE*, 21:739–755, 1991.

[105] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: a survey. *Software Testing, Verification and Reliability*, 22(2):67–120, 2012.

[106] A. Zeller. *Why programs fail: a guide to systematic debugging*. Elsevier, 2009.

[107] K. Zhang and D. Shasha. Simple fast algorithms for the editing distance between trees and related problems. *SIAM journal on computing*, 18(6):1245–1262, 1989.

[108] T. Zhang, M. Song, J. Pinedo, and M. Kim. Interactive code review for systematic changes. In *37th IEEE/ACM International Conference on Software Engineering, ICSE 2015, Volume 1*, pages 111–122, 2015.

Appendix A

ChgCutter: An Intermediate Version Generation Tool

The appendix presents ChgCutter features with a motivating example from JFreeChart project, an information visualization library to display graphs and charts. We adapt and simply the example for the presentation purpose. Suppose David updates a program with two independent development tasks: (1) refactorings by standard refactoring techniques and (2) bug-fixes to resolve some graphical rendering issues. Then he commits his changes in one single transaction to a version control repository with a message, *"Updated several methods by applying the refactorings. Also, fixed graphical rendering bugs by adding null checkers."*.

Suppose Monica conducts a code review for refacotring changes. She first selects and *Compare with Each Other* between *jfreechat_demo_new* and *jfreechat_demo_old* projects from *Project Explorer* as Figure A.1.

Eclipse Compare View. All files different between old and new versions are displayed in Eclipse Compare View (see ① in Figure A.2). Monica selects one changed file, *AbstractRenderer.java*, to inspect differences (see ② in Figure A.2). Locations having changes are shown in *Java Source Compare View*. Monica highlights a sub-region of a diff patch which is one of refactoring changes in *lookupSeriesPaint* method (see ③ in Figure A.2).

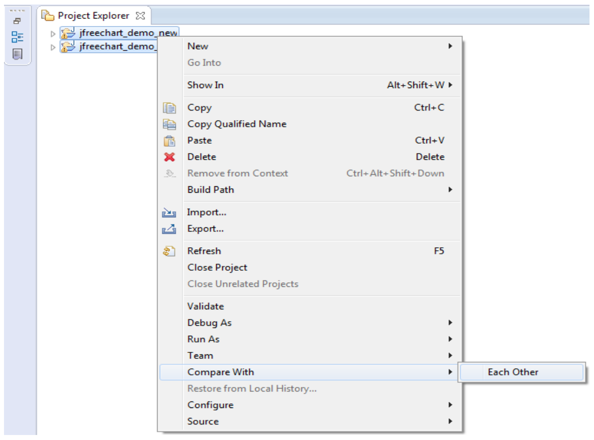
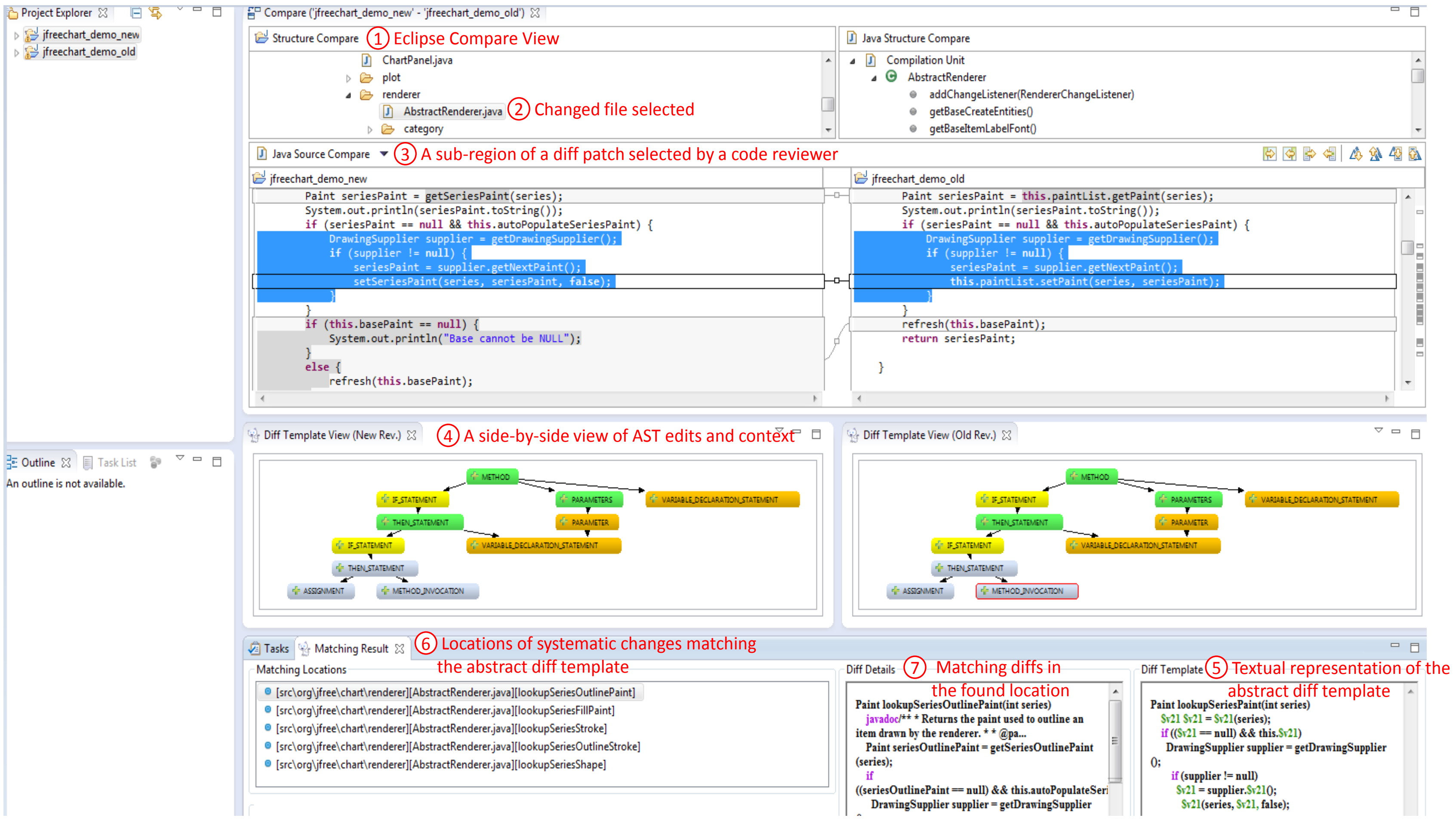


Figure A.1: A screen snapshot of *Eclipse Compare With Each Other* for two projects

Diff Template View. Monica clicks "*Select Diff Region*" in menu list in Figure A.3. Given Monica's selected change region, ChgCutter parses code fragments into AST representation. The data and control dependency analysis is performed to extract dependent context. ChgCutter visualizes dependent context as abstract diff template (see ④ in Figure A.2). Green nodes represent the selected codes' parent nodes. Yellow nodes represent change control-dependant statements. Orange nodes represent change data-dependant statements. To increase a size of change subsets, Monica can review and generalize identifiers into parameters that can be equivalent to different identifiers during matching analysis. The textual template can be previewed in *Diff Template* (see ⑤ in Figure A.2). **Matching Result View.** Monica clicks "*Summarize Changes*" in menu list in Figure A.4. Locations having systematic changes matching the abstract diff template are listed in Matching Locations View (see ⑥ in Figure A.2). When Monica clicks one of matching locations, the corresponding differences are presented in the Diff Details View (see ⑦ in Figure A.2).

Intermediate Version Project. Monica clicks "*Create Intermediate Version*" in menu list in Figure A.5. An intermediate version, *jfreechat_demo_old_iv1*, is automatically created by applying two Update Variable Declaration Statement Operations, in which `Paint seriesPaint = this.paintList.getPaint(series)` is updated by `Paint seriesPaint = getSeriesPaint(series)` and `this.paintList.setPaint(series, seriesPaint)` is updated by `setSeriesPaint(series, seriesPaint, false)` (see ① in Figure A.6). When comparing old and intermediate version using Eclipse Compare, only 6 refactoring changes in the different methods are applied to old version (see ② and ③ in Figure A.6).

Similarly, Figure A.7 and Figure A.8 show the procedure to create Bug-fix intermediate version, *jfreechat_demo_old_iv2*, if Monica is interested in bug-fix changes.



jfreechart_demo_new

```
        Paint seriesPaint = getSeriesPaint(series);
        System.out.println(seriesPaint.toString());
        if (seriesPaint == null && this.autoPopulateSeriesPaint) {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                seriesPaint = supplier.getNextPaint();
                setSeriesPaint(series, seriesPaint, false);
            }
        }
        if (this.basePaint == null) {
            System.out.println("Base cannot be NULL");
        }
        else {
            refresh(this.basePaint);
```

jfreechart_demo_old

```
        Paint seriesPaint = this.paintList.getPaint(series);
        System.out.println(seriesPaint.toString());
        if (seriesPaint == null && this.autoPopulateSeriesPaint) {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                seriesPaint = supplier.getNextPaint();
                this.paintList.setPaint(series, seriesPaint);
            }
        }
        refresh(this.basePaint);
        return seriesPaint;

    }
```

Matching Locations

- [src\org\jfree\chart\renderer][AbstractRenderer.java][lookupSeriesOutlinePaint]
- [src\org\jfree\chart\renderer][AbstractRenderer.java][lookupSeriesFillPaint]
- [src\org\jfree\chart\renderer][AbstractRenderer.java][lookupSeriesStroke]
- [src\org\jfree\chart\renderer][AbstractRenderer.java][lookupSeriesOutlineStroke]
- [src\org\jfree\chart\renderer][AbstractRenderer.java][lookupSeriesShape]

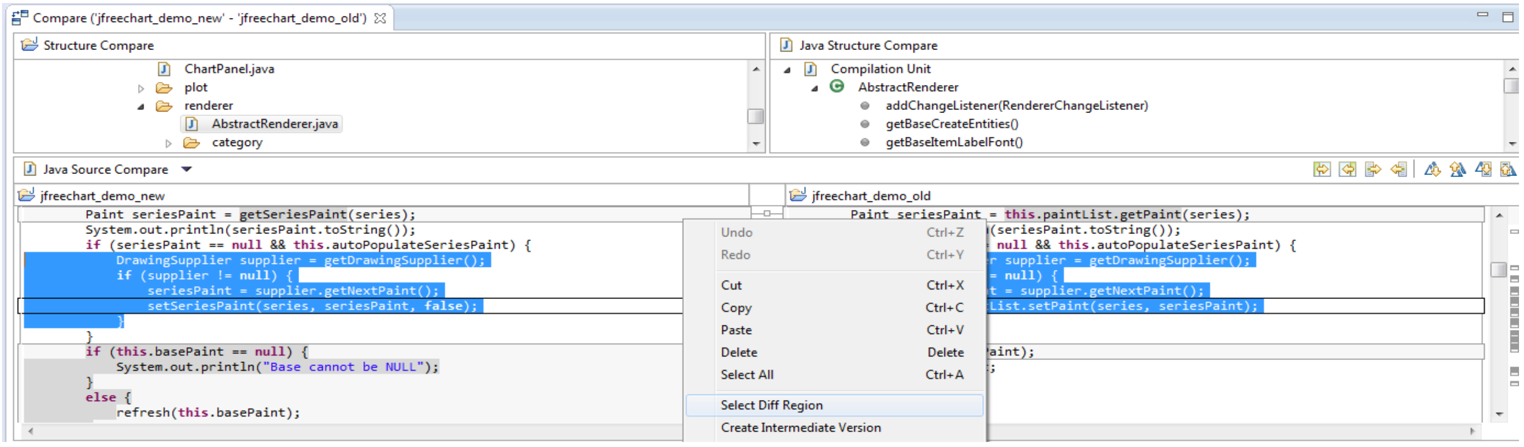
Diff Details

```
Paint lookupSeriesOutlinePaint(int series)
  javadoc/** * Returns the paint used to outline an
item drawn by the renderer. * * @pa...
  Paint seriesOutlinePaint = getSeriesOutlinePaint
(series);
  if
((seriesOutlinePaint == null) && this.autoPopulateSeri
    DrawingSupplier supplier = getDrawingSupplier
```

Diff Template

```
Paint lookupSeriesPaint(int series)
  $v21 $v21 = $v21(series);
  if (($v21 == null) && this.$v21)
   DrawingSupplier supplier = getDrawingSupplier
();
     if (supplier != null)
      $v21 = supplier.$v21();
       $v21(series, $v21, false);
```

Figure A.2: A screen snapshot of ChgCutter to find matching locations for refactoring changes



```
        Paint seriesPaint = getSeriesPaint(series);
        System.out.println(seriesPaint.toString());
        if (seriesPaint == null && this.autoPopulateSeriesPaint) {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                seriesPaint = supplier.getNextPaint();
                setSeriesPaint(series, seriesPaint, false);
            }
        }
        if (this.basePaint == null) {
            System.out.println("Base cannot be NULL");
        }
        else {
            refresh(this.basePaint);
```

```
        Paint seriesPaint = this.paintList.getPaint(series);
                           (seriesPaint.toString());
                           null && this.autoPopulateSeriesPaint) {
                           r supplier = getDrawingSupplier();
                           = null) {
                           t = supplier.getNextPaint();
                           List.setPaint(series, seriesPaint);

                           aint);
                           ;
```

Figure A.3: A screen snapshot of *Select Diff Region*

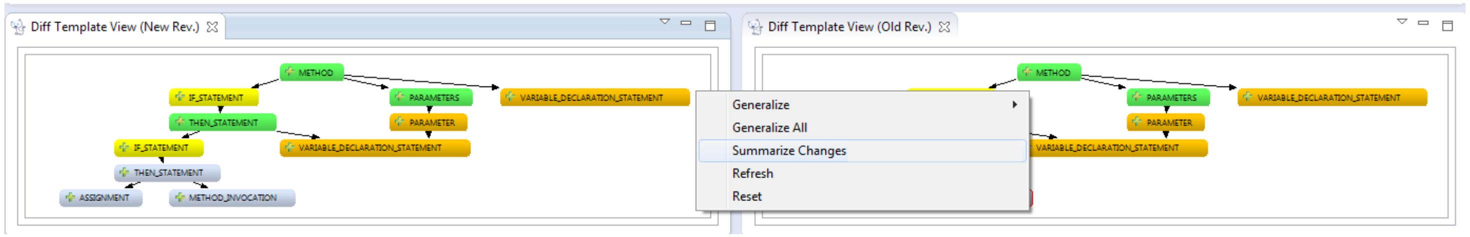
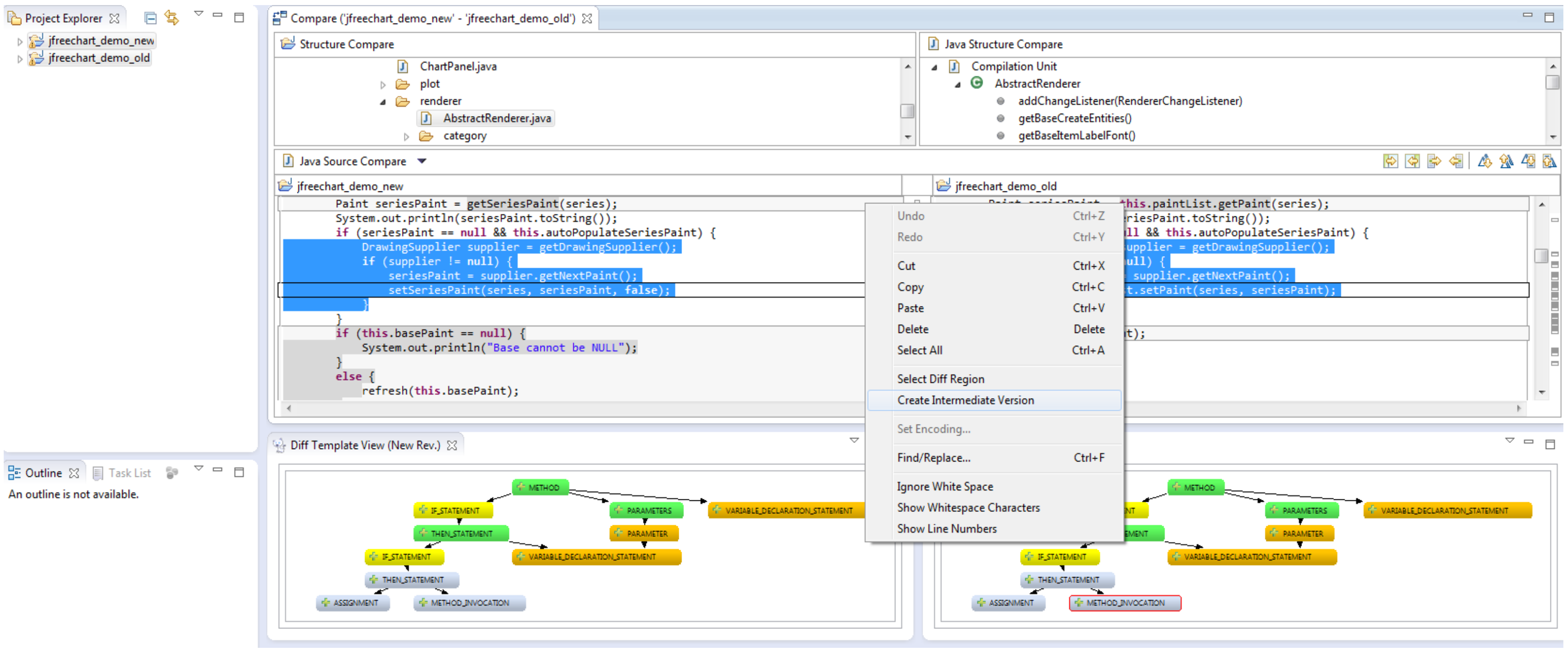


Figure A.4: A screen snapshot of *Summarize Changes*

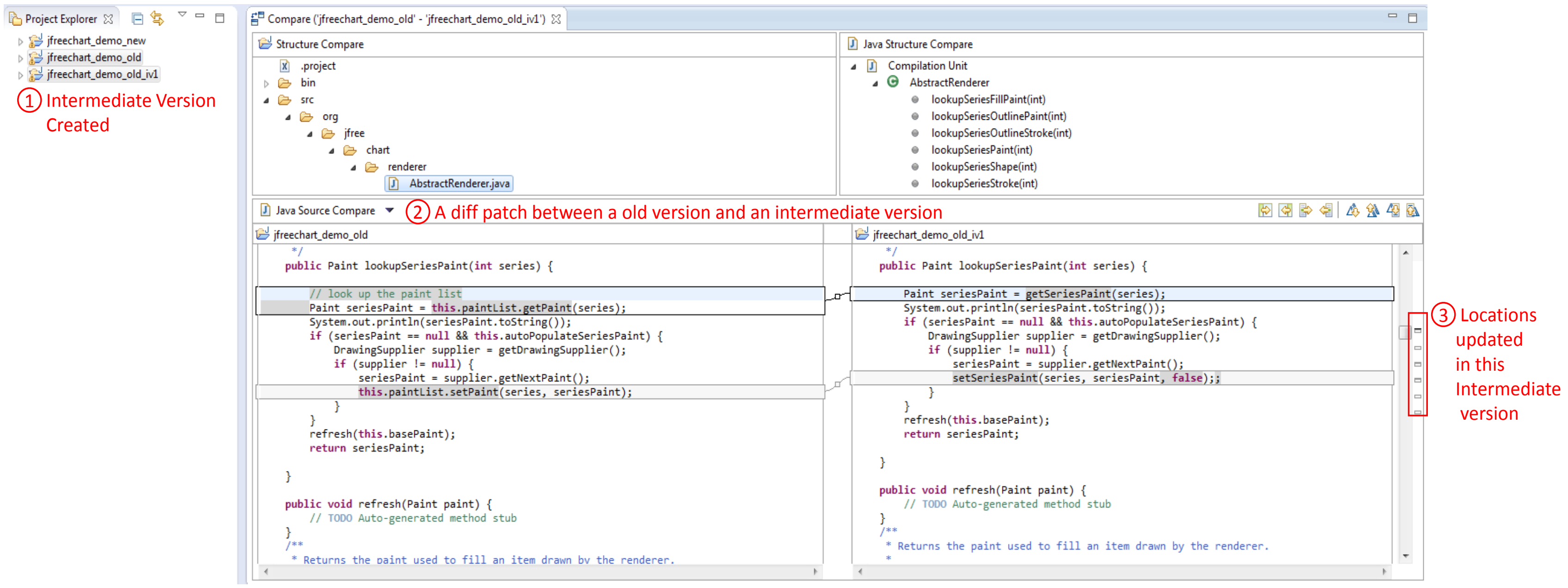


```
Paint seriesPaint = getSeriesPaint(series);
System.out.println(seriesPaint.toString());
if (seriesPaint == null && this.autoPopulateSeriesPaint) {
    DrawingSupplier supplier = getDrawingSupplier();
    if (supplier != null) {
        seriesPaint = supplier.getNextPaint();
        setSeriesPaint(series, seriesPaint, false);
    }
}
if (this.basePaint == null) {
    System.out.println("Base cannot be NULL");
}
else {
    refresh(this.basePaint);
```

```
this.paintList.getPaint(series);
riesPaint.toString());
ll && this.autoPopulateSeriesPaint) {
upplier = getDrawingSupplier();
ull) {
 supplier.getNextPaint();
t.setPaint(series, seriesPaint);

t);
```

Figure A.5: A screen snapshot of *Create Intermediate Version*



jfreechart_demo_old

```java
     */
    public Paint lookupSeriesPaint(int series) {

        // look up the paint list
        Paint seriesPaint = this.paintList.getPaint(series);
        System.out.println(seriesPaint.toString());
        if (seriesPaint == null && this.autoPopulateSeriesPaint) {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                seriesPaint = supplier.getNextPaint();
                this.paintList.setPaint(series, seriesPaint);
            }
        }
        refresh(this.basePaint);
        return seriesPaint;

    }

    public void refresh(Paint paint) {
        // TODO Auto-generated method stub
    }
    /**
     * Returns the paint used to fill an item drawn by the renderer.
```

jfreechart_demo_old_iv1

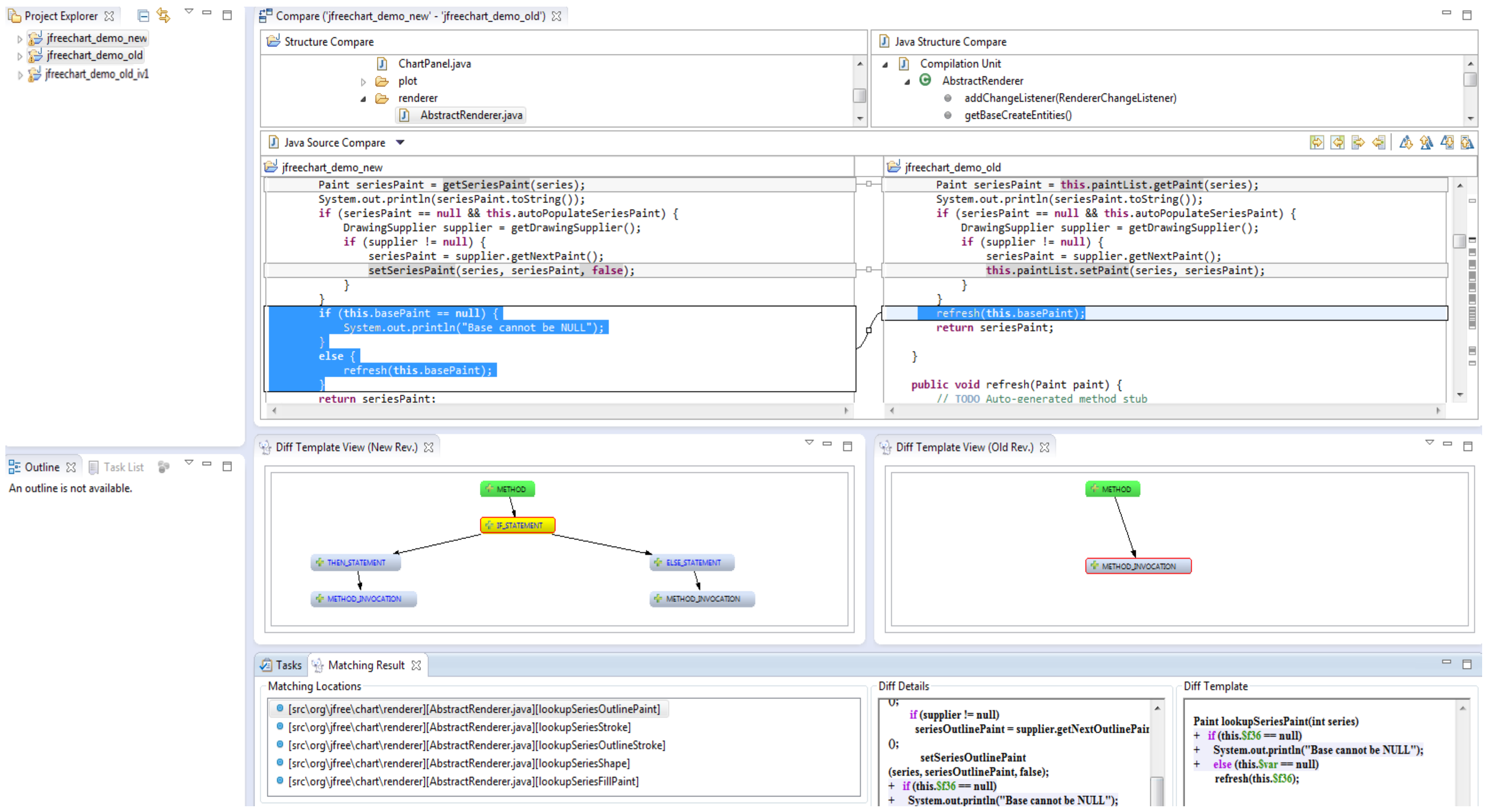
```java
     */
    public Paint lookupSeriesPaint(int series) {

        Paint seriesPaint = getSeriesPaint(series);
        System.out.println(seriesPaint.toString());
        if (seriesPaint == null && this.autoPopulateSeriesPaint) {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                seriesPaint = supplier.getNextPaint();
                setSeriesPaint(series, seriesPaint, false);;
            }
        }
        refresh(this.basePaint);
        return seriesPaint;

    }

    public void refresh(Paint paint) {
        // TODO Auto-generated method stub
    }
    /**
     * Returns the paint used to fill an item drawn by the renderer.
     *
```

Figure A.6: A screen snapshot of CHGCUTTER to create an intermediate version for refactoring changes



jfreechart_demo_new

```
        Paint seriesPaint = getSeriesPaint(series);
        System.out.println(seriesPaint.toString());
        if (seriesPaint == null && this.autoPopulateSeriesPaint) {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                seriesPaint = supplier.getNextPaint();
                setSeriesPaint(series, seriesPaint, false);
            }
        }
        if (this.basePaint == null) {
            System.out.println("Base cannot be NULL");
        }
        else {
            refresh(this.basePaint);
        }
        return seriesPaint:
```

jfreechart_demo_old

```
        Paint seriesPaint = this.paintList.getPaint(series);
        System.out.println(seriesPaint.toString());
        if (seriesPaint == null && this.autoPopulateSeriesPaint) {
            DrawingSupplier supplier = getDrawingSupplier();
            if (supplier != null) {
                seriesPaint = supplier.getNextPaint();
                this.paintList.setPaint(series, seriesPaint);
            }
        }
        refresh(this.basePaint);
        return seriesPaint;

    }

    public void refresh(Paint paint) {
        // TODO Auto-generated method stub
```

Matching Locations

```
[src\org\jfree\chart\renderer][AbstractRenderer.java][lookupSeriesOutlinePaint]
[src\org\jfree\chart\renderer][AbstractRenderer.java][lookupSeriesStroke]
[src\org\jfree\chart\renderer][AbstractRenderer.java][lookupSeriesOutlineStroke]
[src\org\jfree\chart\renderer][AbstractRenderer.java][lookupSeriesShape]
[src\org\jfree\chart\renderer][AbstractRenderer.java][lookupSeriesFillPaint]
```

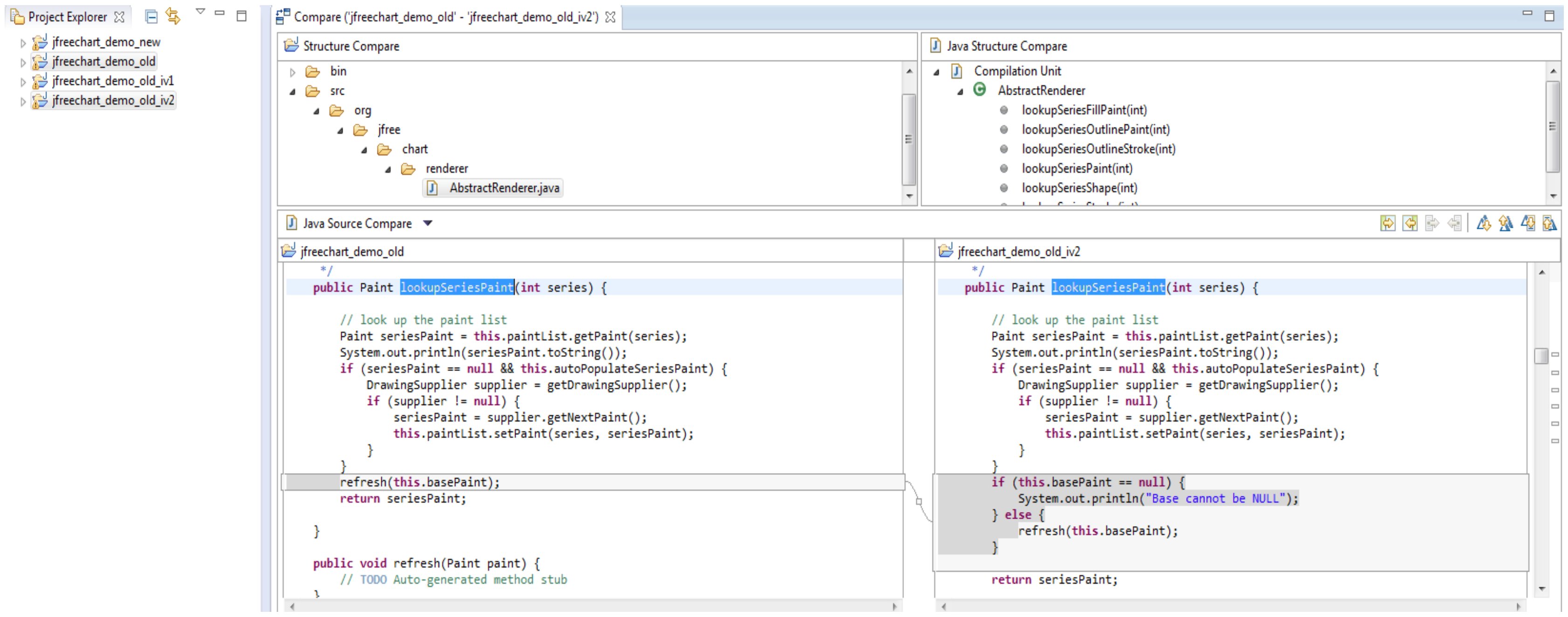
Diff Details

```
0;
    if (supplier != null)
     seriesOutlinePaint = supplier.getNextOutlinePair
0;
      setSeriesOutlinePaint
(series, seriesOutlinePaint, false);
+  if (this.$f36 == null)
+    System.out.println("Base cannot be NULL");
```

Diff Template

```
Paint lookupSeriesPaint(int series)
+  if (this.$f36 == null)
+    System.out.println("Base cannot be NULL");
+    else (this.$var == null)
       refresh(this.$f36);
```

Figure A.7: A screen snapshot of CHGCUTTER to find matching locations for Bug-fix changes



```
    */
public Paint lookupSeriesPaint(int series) {

    // look up the paint list
    Paint seriesPaint = this.paintList.getPaint(series);
    System.out.println(seriesPaint.toString());
    if (seriesPaint == null && this.autoPopulateSeriesPaint) {
        DrawingSupplier supplier = getDrawingSupplier();
        if (supplier != null) {
            seriesPaint = supplier.getNextPaint();
            this.paintList.setPaint(series, seriesPaint);
        }
    }
    refresh(this.basePaint);
    return seriesPaint;

}

public void refresh(Paint paint) {
    // TODO Auto-generated method stub
}
```

```
    */
public Paint lookupSeriesPaint(int series) {

    // look up the paint list
    Paint seriesPaint = this.paintList.getPaint(series);
    System.out.println(seriesPaint.toString());
    if (seriesPaint == null && this.autoPopulateSeriesPaint) {
        DrawingSupplier supplier = getDrawingSupplier();
        if (supplier != null) {
            seriesPaint = supplier.getNextPaint();
            this.paintList.setPaint(series, seriesPaint);
        }
    }
    if (this.basePaint == null) {
        System.out.println("Base cannot be NULL");
    } else {
        refresh(this.basePaint);
    }

    return seriesPaint;
```

Figure A.8: A screen snapshot of ChgCutter to create an intermediate version for Bug-fix changes