Improving Effectiveness of Automated Software Testing in the Absence of Specifications

Tao Xie

A dissertation submitted in partial fulfillment
of the requirements for the degree of

Doctor of Philosophy

University of Washington

2005

Program Authorized to Offer Degree: Computer Science and Engineering

University of Washington
Graduate School

This is to certify that I have examined this copy of a doctoral dissertation by

Tao Xie

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Chair of Supervisory Committee:

_____

David Notkin

Reading Committee:

_____

Richard Anderson

_____

David Notkin

_____

Wolfram Schulte

Date: _____

In presenting this dissertation in partial fulfillment of the requirements for the doctoral degree at the University of Washington, I agree that the Library shall make its copies freely available for inspection. I further agree that extensive copying of this dissertation is allowable only for scholarly purposes, consistent with “fair use” as prescribed in the U.S. Copyright Law. Requests for copying or reproduction of this dissertation may be referred to Proquest Information and Learning, 300 North Zeeb Road, Ann Arbor, MI 48106-1346, to whom the author has granted “the right to reproduce and sell (a) copies of the manuscript in microform and/or (b) printed copies of the manuscript made from microform.”

Signature_____

Date_____

University of Washington

Abstract

Improving Effectiveness of Automated Software Testing in the Absence of Specifications

Tao Xie

Chair of Supervisory Committee:

Professor David Notkin
Computer Science and Engineering

This dissertation presents techniques for improving effectiveness of automated software testing in the absence of specifications, evaluates the efficacy of these techniques, and proposes directions for future research.

Software testing is currently the most widely used method for detecting software failures. When testing a program, developers need to generate test inputs for the program, run these test inputs on the program, and check the test execution for correctness. It has been well recognized that software testing is quite expensive, and automated software testing is important for reducing the laborious human effort in testing. There are at least two major technical challenges in automated testing: the generation of sufficient test inputs and the checking of the test execution for correctness. Program specifications can be valuable in addressing these two challenges. Unfortunately, specifications are often absent from programs in practice.

This dissertation presents a framework for improving effectiveness of automated testing in the absence of specifications. The framework supports a set of related techniques. First, it includes a redundant-test detector for detecting redundant tests among automatically generated test inputs. These redundant tests increase testing time without increasing the ability to detect faults or increasing our confidence in the program. Second, the framework includes a non-redundant-test generator that employs state-exploration techniques to generate non-redundant tests in the first place and uses

symbolic execution techniques to further improve the effectiveness of test generation. Third, because it is infeasible for developers to inspect the execution of a large number of generated test inputs, the framework includes a test selector that selects a small subset of test inputs for inspection; these selected test inputs exercise new program behavior that has not been exercised by manually created tests. Fourth, the framework includes a test abstractor that produces succinct state transition diagrams for inspection; these diagrams abstract and summarize the behavior exercised by the generated test inputs. Finally, the framework includes a program-spectra comparator that compares the internal program behavior exercised by regression tests executed on two program versions, exposing behavioral differences beyond different program outputs.

The framework has been implemented and empirical results have shown that the developed techniques within the framework improve the effectiveness of automated testing by detecting high percentage of redundant tests among test inputs generated by existing tools, generating non-redundant test inputs to achieve high structural coverage, reducing inspection efforts for detecting problems in the program, and exposing more behavioral differences during regression testing.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

Portions of this dissertation were previously published at ASE 04 [XMN04b] (Chapter 3), TACAS 05 [XMSN05] (Chapter 4), ASE 03 [XN03b] (Chapter 5), ICFEM 04 [XN04a] (Chapter 6), ICSM 04 [XN04c] (Chapter 7), and FATES 03 [XN03a] (Section 2.6). The material included in this dissertation has been updated and extended. I was fortunate to have Darko Marinov, David Notkin, and Wolfram Schulte as collaborators in my dissertation research. I would like to thank them for their help with the research. I also would like to thank my Ph.D. Supervisory Committee (Richard Anderson, Craig Chambers, David Notkin, and Wolfram Schulte) for valuable guidance on both research and writing.

I especially would like to thank my advisor, David Notkin, for supporting me with freedom and guidance throughout my Ph.D. studies. David has provided me valuable technical and professional advice, which will continue to benefit me in my future career. I am also grateful to my several long-distance research collaborators. Darko Marinov has made our collaborative work (on redundant-test detection and test generation) productive and enjoyable; Wolfram Schulte has provided his insightful perspective during our collaborative work (on test generation); Jianjun Zhao has helped extend my existing research to testing AspectJ programs; Collaborating with me on the Stabilizer project [MX05], Amir Michail has helped expand my research to a broader scope.

I want to thank the people who provided tools and experimental subjects used in my research. Michael Ernst and his research group provided the Daikon tool and some experimental subjects. Mary Jean Harrold and Gregg Rothermel provided the siemens programs. Darko Marinov provided the Korat benchmarks and the Ferastrau mutation testing tool. Yu-Seung Ma and Jeff Offutt provided the Jmutation mutation testing tool. Christoph Csallner and Yannis Smaragdakis provided the JCrasher testing tool. Corina Pasareanu and Willem Visser provided the Java Pathfinder tool. Alex Kanevsky, Roman Salvador, and Roberto Scaramuzzi from Parasoft helped provide the Jtest tool. Atif Memon and Qing Xie provided the TerpOffice benchmarks (used in the Stabilizer

project [MX05]). Alexandru Salcianu provided the static side-effect-analysis tool (used in detecting redundant tests for AspectJ programs [XZMN04]).

Many colleagues kindly offered their helps in shaping and improving my research. I want to thank Jonathan Aldrich, Andrei Alexandrescu, Richard Anderson, Michael Barnett, Brian Bershad, Craig Chambers, Christoph Csallner, Yuetang Deng, Roongko Doong, Sebastian Elbaum, Michael Ernst, David Evans, Birgit Geppert, Arnaud Gotlieb, Wolfgang Grieskamp, Neelam Gupta, Mary Jean Harrold, Susan Horwitz, Sarfraz Khurshid, Miryung Kim, Robert Lai, Keunwoo Lee, Bruno Legeard, Yu Lei, Sorin Lerner, Darko Marinov, Atif Memon, Amir Michail, Todd Millstein, Nick Mitchell, David Notkin, Jeff Offutt, Alex Orso, Corina Pasareanu, Andrew Petersen, Will Portnoy, Gregg Rothermel, Alberto Savoia, Vibha Sazawal, Wolfram Schulte, Gary Sevitsky, Yannis Smaragdakis, Kevin Sullivan, Nikolai Tillmann, Willem Visser, David Weiss, Michal Young, Andreas Zeller, and Jianjun Zhao.

The anonymous reviewers for FATES 03, ASE 03, ASE 04, ICSM 04, ICFEM 04, TACAS 05, ASE journal, and IEEE TSE journal provided valuable feedback to help improve my papers that this dissertation is based on. This research was supported in part by NSF grant ITR 0086003 and in part by the High Dependability Computing Program from NASA Ames cooperative agreement NCC-2-1298.

I would like to thank Darko Marinov, Atif Memon, David Notkin, Wolfram Schulte, and David Weiss, who wrote recommendation letters for me and helped me throughout my job search process. I also want to thank the people from different schools who helped my job search. I would like to thank the faculty at North Carolina State University for offering me a job.

Last but not least, I am grateful to my friends and family. My parents Xinmin Xie and Zhijian He have encouraged me and believed in me for many years. My brother Yuan Xie and sister-in-law Hong Zeng have given me continuous support.

Chapter 1

INTRODUCTION

Software permeates many aspects of our life; thus, improving software reliability is becoming critical to society. A recent report by National Institute of Standards and Technology found that software errors cost the U.S. economy about $60 billion each year [NIS02]. Although much progress has been made in software verification and validation, software testing is still the most widely used method for improving software reliability. However, software testing is labor intensive, typically accounting for about half of the software development effort [Bei90].

To reduce the laborious human effort in testing, developers can conduct automated software testing by using tools to automate some activities in software testing. Software testing activities typically include generating test inputs, creating expected outputs, running test inputs, and verifying actual outputs. Developers can use some existing frameworks or tools such as the JUnit testing framework [GB03] to write unit-test inputs and their expected outputs. Then the JUnit framework can automate running test inputs and verifying actual outputs against the expected outputs. To reduce the burden of manually creating test inputs, developers can use some existing test-input generation tools [Par03, CS04, Agi04] to generate test inputs automatically. After developers modify a program, they can conduct *regression testing* by rerunning the existing test inputs in order to assure that no regression faults are introduced. Even when expected outputs are not created for the existing test inputs, the actual outputs produced by the new version can be automatically compared with the ones produced by the old version in order to detect behavioral differences.

However, the existing test-generation tools often cannot effectively generate sufficient test inputs to expose program faults or increase code coverage. In addition, when these tools are used to generate test inputs automatically, expected outputs for these test inputs are still missing, and it is infeasible for developers to create expected outputs for this large number of generated test inputs.

Although specifications can be used to improve the effectiveness of generating test inputs and check program correctness when running test inputs without expected outputs, specifications often do not exist in practice. In regression testing, the existing approach of comparing observable outputs is limited in exposing behavioral differences inside program execution; these differences could be symptoms of potential regression faults.

Our research focuses on developing a framework for improving effectiveness of automated testing in the absence of specifications. The framework includes techniques and tools for improving the effectiveness of generating test inputs and inspecting their executions for correctness, two major challenges in automated testing.

This chapter discusses activities and challenges of automated software testing (Section 1.1), lists the contributions of the dissertation: a framework for improving effectiveness of automated testing (Section 1.2), defines the scope of the research in the dissertation (Section 1.3), and gives an organization of the remainder of the dissertation (Section 1.4).

1.1 Activities and Challenges of Automated Software Testing

Software testing activities consist of four main steps in testing a program: generating test inputs, generating expected outputs for test inputs, run test inputs, and verify actual outputs. To reduce the laborious human effort in these testing activities, developers can automate these activities to some extent by using testing tools. Our research focuses on developing techniques and tools for addressing challenges of automating three major testing activities: generating test inputs, generating expected outputs, and verifying actual outputs, particularly in the absence of specifications, because specifications often do not exist in practice. The activities and challenges of automated software testing are described below.

Generate (*sufficient*) test inputs. Test-input generation (in short, test generation) often occurs when an implementation of the program under test is available. However, before a program implementation is available, test inputs can also be generated automatically during model-based test generation [DF93, GGSV02] or manually during test-driven development [Bec03], a key practice of Extreme Programming [Bec00]. Because generating test inputs manually is often labor intensive, developers can use test-generation tools [Par03, CS04, Agi04] to generate test

inputs automatically or use measurement tools [Qui03, JCo03, Hor02] to help developers determine where to focus their efforts. Test inputs can be constructed based on the program's specifications, code structure, or both. For an object-oriented program such as a Java class, a test input typically consists of a sequence of method calls on the objects of the class.

Although the research on automated test generation is more than three decades old [Hua75, Kin76, Cla76, RHC76], automatically generating sufficient test inputs still remains a challenging task. Early work as well as some recent work [Kor90, DO91, KAY96, GMS98, GBR98, BCM04] primarily focuses on procedural programs such as C programs. More recent research [KSGH94, BOP00, Ton04, MK01, BKM02, KPV03, VPK04] also focuses on generating test inputs for object-oriented programs, which are increasingly pervasive. Generating test inputs for object-oriented programs adds additional challenges, because inputs for method calls consist of not only method arguments but also receiver-object states, which are sometimes *structurally complex* inputs, such as linked data structures that must satisfy complex properties. Directly constructing receiver-object states requires either dedicated algorithms [BHR+00] or class invariants [LBR98, LG00] for specifying properties satisfied by valid object states; however, these dedicated algorithms or class invariants are often not readily available in part because they are difficult to write. Alternatively, method sequences can be generated to construct desired object states indirectly [BOP00, Ton04]; however, it is generally expensive to enumerate all possible method sequences even given a small number of argument values and a small bound on the maximum sequence length.

Generate expected outputs (for a *large* number of test inputs). Expected outputs are generated to help determine whether the program behaves correctly on a particular execution during testing. Developers can generate an expected output for each specific test input to form precomputed input/output pair [Pan78, Ham77]. For example, the JUnit testing framework [GB03] allows developers to write assertions in test code for specifying expected outputs. Developers can also write checkable specifications [Bei90, BGM91, DF93, MK01, CL02, BKM02, GGSV02] for the program and these specifications offer expected outputs (more precisely, expected properties) for any test input executed on the program.

It is tedious for developers to generate expected outputs for a large number of test inputs. Even if developers are willing to invest initial effort in generating expected outputs, it is expensive to maintain these expected outputs when the program is changed and some of these expected outputs need to be updated [KBP02, MS03].

Run test inputs (*continuously* and *efficiently*). Some testing frameworks such as the JUnit testing framework [GB03] allow developers to structure several *test cases* (each of which comprises a test input and its expected output) into a *test suite*, and provide tools to run a test suite automatically. For graphical user interface (GUI) applications, running test inputs especially requires dedicated testing frameworks [OAFG98, Mem01, Rob03, Abb04].

In software maintenance, it is important to run regression tests frequently in order to make sure that new program changes do not break the program. Developers can manually start the execution of regression tests after having changed the program or configure to continuously run regression tests in the background while changing the program [SE03]. Sometimes running regression tests is expensive; then developers can use mock objects [MFC01, SE04] to avoid rerunning the parts of the program that are slow and expensive to run. Developers can also use regression test selection [RH97, GHK+01, HJL+01] to select a subset of regression tests to rerun or regression test prioritization [WHLB97, RUCH01, EMR02] to sort regression tests to rerun. Although some techniques proposed in our research can be used to address some challenges in running test inputs, our research primarily addresses the challenges in the other three steps.

Verify actual outputs (in the *absence* of expected outputs). A *test oracle* is a mechanism for checking whether the actual outputs of the program under test is equivalent to the expected outputs [RAO92, Ric94, Hof98, MPS00, BY01]. When expected outputs are unspecified or specified but in a way that does not allow automated checking, the oracle often relies on developers' eyeball inspection. If expected outputs are directly written as executable assertions [And79, Ros92] or translated into runtime checking code [GMH81, Mey88, MK01, CL02, BKM02, GGSV02], verifying actual outputs can be automated. When no expected outputs are available, developers often rely on program crashes [MFS90, KJS98] or uncaught excep-

tions [CS04] as symptoms for unexpected behavior. When no expected outputs are specified explicitly, in regression testing, developers can compare the actual outputs of a new version of the program with the actual outputs of a previous version [Cha82].

As has been discussed in the second step, it is challenging to generate expected outputs for a large number of test inputs. In practice, expected outputs often do not exist for automatically generated test inputs. Without expected outputs, it is often expensive and prone to error for developers to manually verify the actual outputs and it is limited in exploiting these generated test inputs by verifying only whether the program crashes [MFS90,KJS98] or throws uncaught exceptions [CS04]. In regression testing, the actual outputs of a new version can be compared with the actual outputs of its previous version. However, behavioral differences between versions often cannot be propagated to the observable outputs that are compared between versions.

A *test adequacy criterion* is a condition that an adequate test suite must satisfy in exercising a program's properties [GG75]. Common criteria [Bei90] include structural coverage: code coverage (such as statement, branch, or path coverage) and specification coverage [CR99]. Coverage measurement tools can be used to evaluate a test suite against a test adequacy criterion automatically. A test adequacy criterion provides a stopping rule for testing (a rule to determine whether sufficient testing has been performed and it can be stopped) and a measurement of test-suite quality (a degree of adequacy associated with a test suite) [ZHM97]. A test adequacy criterion can be used to guide the above four testing activities. For example, it can be used to help determine what test inputs are to be generated and which generated test inputs are to be selected so that developers can invest efforts in equipping the selected inputs with expected outputs, run these inputs, and verify their actual outputs. After conducting these four activities, a test adequacy criterion can be used to determine if the program has been adequately tested and to further identify which parts of the program have not been adequately tested.

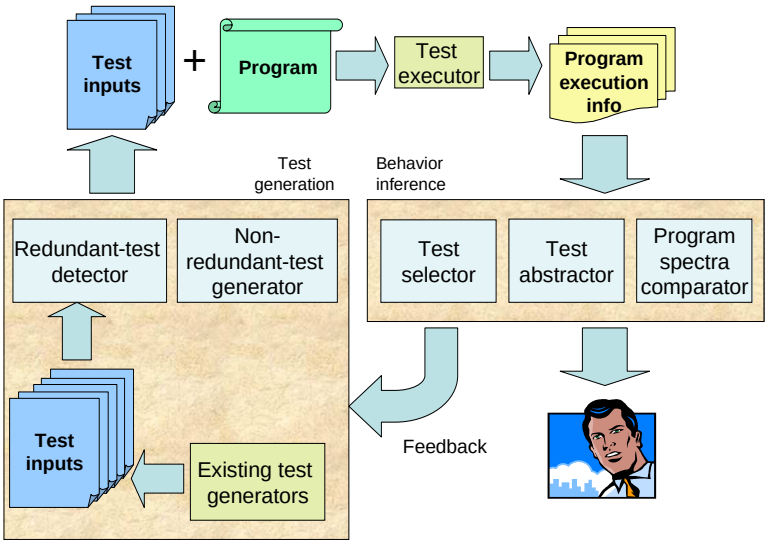

Figure 1.1: Framework for improving effectiveness of automated testing.

1.2 Contributions

This dissertation presents a framework for improving effectiveness of automated testing, addressing the challenges discussed in the preceding section. As is shown in Figure 1.1, the framework consists of two groups of components. The first group of components —the redundant-test detector and non-redundant-test generator— address the issues in generating test inputs. The second group of components (the test selector, test abstractor, and program-spectra comparator) infer program behavior dynamically in order to address the issues in checking the correctness of test executions. The second group of components further send feedback information to the first group to guide test generation.

Redundant-test detector. Existing test generation tools generate a large number of test inputs[1] (in short, tests) to exercise different sequences of method calls in the interface of the class under test. Different combinations of method calls on the class under test result in a combinatorial explosion of tests. Because of resource constraints, existing test generation tools often generate different sequences of method calls whose lengths range from one [CS04] to three [Par03].

[1] In the rest of the dissertation, we use *tests* to denote *test inputs* for the sake of simplicity.

However, sequences of up-to-three method calls are often insufficient for detecting faults or satisfying test adequacy criteria. In fact, a large portion of these different sequences of method calls exercise no new method behavior; in other words, the tests formed by this large portion of sequences are *redundant tests*. We have defined redundant tests by using method-input values (including both argument values and receiver-object states). When the method-input values of each method call in a test have been exercised by the existing tests, the test is considered as a redundant test even if the sequence of method calls in the test is different from the one of any existing test. We have developed a redundant-test detector, which can post-process a test suite generated by existing test generation tools and output a reduced test suite containing no redundant tests. Our approach not only presents a foundation for existing tools that generate non-redundant tests [MK01, BKM02, KPV03, VPK04] but also enables any other test generation tools [Par03, CS04, Agi04] to avoid generating redundant tests by incorporating the redundant-test detection in their test generation process. We present experimental results that show the effectiveness of the redundant-test detection tool: about 90% of the tests generated by a commercial testing tool [Par03] are detected and reduced by our tool as redundant tests.

Non-redundant-test generator. Based on the notion of avoiding generating redundant-tests, we have developed a non-redundant-test generator, which explores the concrete or symbolic receiver-object state space by using method calls (through normal program execution or symbolic execution). Like some other software model checking tools based on stateful exploration [DIS99, VHBP00, CDH$^+$00, MPC$^+$02, RDH03], the test generator based on concrete-state exploration faces the state explosion problem. Symbolic representations in symbolic model checking [McM93] alleviate the problem by describing not only single states but sets of states; however, existing software model checking tools [BR01, HJMS03] based on symbolic representations are limited for handling complex data structures. Recently, symbolic execution [Kin76, Cla76] has been used to directly construct symbolic states for receiver objects [KPV03, VPK04]; however, the application of symbolic execution requires the user to provide specially constructed class invariants [LG00], which effectively describe an over-approximation of the set of reachable object graphs. Without requiring any class invariant, our test generator can also use symbolic execution of method sequences to explore the symbolic

receiver-object states and prune this exploration based on novel state comparisons (comparing both heap representations and symbolic representations). Our extension and application of symbolic execution in state exploration not only alleviate the state explosion problem but also generate relevant method arguments for method sequences automatically by using a constraint solver [SR02]. We present experimental results that show the effectiveness of the test generation based on symbolic-state exploration: it can achieve higher branch coverage faster than the test generation based on concrete-state exploration.

Test selector. Because it is infeasible for developers to inspect the actual outputs of a large number of generated tests, we have developed a test selector to select a small valuable subset of generated tests for inspection. These selected tests exercise new behavior that has not been exercised by the existing test suite. In particular, we use Daikon [Ern00] to infer program behavior dynamically from the execution of the existing (manually) constructed test suite. We next feed inferred behavior in the form of specifications to an existing specification-based test generation tool [Par03]. The tool generates tests to violate the inferred behavior. These violating tests are selected for inspection, because these violating tests exhibit behavior different from the behavior exhibited by the existing tests. Developers can inspect these violating tests together with the violated properties, equip these tests with expected outputs, and add them to the existing test suite. We present experimental results to show that the selected tests have a high probability of exposing anomalous program behavior (either faults or failures) in the program.

Test abstractor. Instead of selecting a subset of generated tests for inspection, a test abstractor summarizes and abstracts the receiver-object-state transition behavior exercised by all the generated tests. Because the concrete-state transition diagram for receiver objects is too complicated for developers to inspect, the test abstractor uses a state abstraction technique based on the observers in a class interface; these observers are the public methods whose return types are not void. An abstract state for a concrete state is represented by the concrete state's observable behavior, consisting of the return values of observer-method calls on the concrete state. The abstract states and transitions among them are used to construct succinct state tran-

sition diagrams for developers to inspect. We present an evaluation to show that the abstract-state transition diagrams can help discover anomalous behavior, debug exception-throwing behavior, and understand normal behavior in the class interface.

Program-spectra comparator. In regression testing, comparing the actual outputs of two program versions is limited in exposing the internal behavioral differences during the program execution, because internal behavioral differences often cannot be propagated to observable outputs. A program spectrum is used to characterize a program's behavior [RBDL97]. We propose a new class of program spectra, called *value spectra*, to enrich the existing program spectra family, which primarily include structural spectra (such as path spectra [BL96, RBDL97, HRS+00]). Value spectra capture internal program states during a test execution. A *deviation* is the difference between the value of a variable in a new program version and the corresponding one in an old version. We have developed a program-spectra comparator that compares the value spectra from an old version and a new version, and uses the spectra differences to detect behavior deviations in the new version. Furthermore, value spectra differences can be used to locate deviation roots, which are program locations that trigger the behavior deviations. Inspecting value spectra differences can allow developers to determine whether program changes introduce intended behavioral differences or regression faults. We present experimental results to show that comparing value spectra can effectively expose behavioral differences between versions even when their actual outputs are the same, and value spectra differences can be used to locate deviation roots with high accuracy.

Dynamic behavior inference requires a good-quality test suite to infer behavior that is close to what shall be described by a specification (if it is manually constructed). On the other hand, specification-based test generation can help produce a good-quality test suite but requires specifications, which often do not exist in practice. There seems to be a circular dependency between dynamic behavior inference and (specification-based) test generation. To exploit the circular dependency and alleviate the problem, we propose a feedback loop between behavior inference and test generation. The feedback loop starts with an existing test suite (constructed manually or automatically) or some existing program runs. By using one of the behavior-inference components (the test selector, test abstractor, or program-spectra comparator), we first infer behavior based on

the existing test suite or program runs. We then feed inferred behavior to a specification-based test generation tool or a test generation tool that can exploit the inferred behavior to improve its test generation. The new generated tests can be used to infer new behavior. The new behavior can be further used to guide test generation in the subsequent iteration. Iterations terminate until a user-defined maximum iteration number has been reached or no new behavior has been inferred from new tests. We show several instances of the feedback loop in different types of behavior inference. This feedback loop provides a means to producing better tests and better approximated specifications automatically and incrementally. In addition, the by-products of the feedback loop are a set of selected tests for inspection; these selected tests exhibit new behavior that has not been exercised by the existing tests.

1.3 Scope

The approaches presented in this dissertation focus on automated software testing. The activities of automated software testing are not limited to automating the execution of regression tests, for example, by writing them in the JUnit testing framework [GB03] or test scripts [Rob03, Abb04], or by capturing and replaying them with tools [SCFP00]. Our focused activities of automated software testing have been described in Section 1.1.

The approaches presented in this dissertation focus on testing sequential programs but not concurrent programs. Most approaches presented in this dissertation focus on testing a program unit (such as a class) written in modern object-oriented languages (such as Java). But the regression testing approach focuses on testing a system written in procedural languages (such as C). All the approaches assume that the unit or system under test is a closed unit or system and there is a well-defined interface between the unit or system and its environment. The approaches focus on testing functional correctness or program robustness but not other quality attributes such as performance and security. Chapter 8 discusses future directions of extending the approaches to test new types of programs and new types of quality attributes.

1.4 Outline

The remainder of this dissertation is organized as follows.

Chapter 2 introduces the background information of automated software testing and surveys related work. Chapter 3 describes the techniques for detecting redundant tests among automatically generated tests. Chapter 4 further presents the techniques for generating nonredundant tests in the first place. Chapter 5 describes the techniques for selecting a small subset of generated tests for inspection. Chapter 6 introduces the techniques that abstract the behavior of test executions for inspection. Chapter 7 describes the techniques for comparing value spectra in regression testing in order to expose behavioral differences between versions. Chapter 8 presents suggestions for future work. Finally, Chapter 9 concludes with a summary of the contributions and lessons learned.

Chapter 2

BACKGROUND AND RELATED WORK

This chapter presents background information and discusses how our research relates to other projects in software testing. Section 2.1 discusses test adequacy criteria, which usually specify the objectives of testing. Section 2.2 presents existing automated test generation techniques. Section 2.3 describes existing test selection techniques. Section 2.4 reviews existing regression testing techniques. Section 2.5 presents existing techniques in behavior inference, and Section 2.6 discusses existing feedback loops in program analysis.

2.1 Test Adequacy Criteria

A test adequacy criterion provides a stopping rule for testing and a measurement of test-suite quality [ZHM97]. (A test adequacy criterion can be used to guide test selection, which shall be discussed in Section 2.3.) Based on the source of information used to specify testing requirements, Zhu et al. [ZHM97] classified test adequacy criteria into four groups: program-based criteria, specification-based criteria, combined specification- and program-based criteria, and interface-based criteria. *Program-based criteria* specify testing requirements based on whether all the identified features in a program have been fully exercised. Identified features in a program can be statements, branches, paths, or definition-use paths. *Specification-based criteria* specify testing requirements based on whether all the identified features in a specification have been fully exercised. *Combined specification- and program-based criteria* specify testing requirements based on both specification-based criteria and program-based criteria. *Interface-based criteria* specify testing requirements based on only the interface information (such as type and range of program inputs) without referring to any internal features of a specification or program. *Random testing* is often based on interface-based criteria. Specification-based criteria and interface-based criteria are types of *black-box testing*, whereas program-based criteria and combined specification- and program-based criteria

are types of *white-box testing*.

Our testing research in this dissertation mostly relies on method inputs (both receiver-object states and argument values in an object-oriented program) and method outputs (both receiver-object states and return values in an object-oriented program). This is related to interface-based criteria. But our research on test selection and abstraction is performed based on inferred behavior, which is often in the form of specifications; therefore, the research is also related to specification-based criteria (but without requiring specifications). Our research on test generation additionally uses symbolic execution to explore paths within a method; therefore, the research is also related to program-based criteria.

In particular, our testing research is related to program-based test adequacy criteria proposed to operate in the semantic domain of program properties rather than the syntactic domain of program text, which is the traditional focus of most program-based criteria. Hamlet's probable correctness theory [Ham87] suggests *data-coverage testing* to uniformly sample the possible values of all internal variables at each control point in a program. However, it is often difficult or undecidable to determine the possible values for variables in a program; therefore, we cannot compute the goal of 100 percent coverage (denominator) for data coverage criteria like for code coverage criteria (such as statement or branch coverage) but use the data coverage of a given test suite as a baseline for comparison. Harder et al. [HME03] use operational abstractions [ECGN01] inferred from a test suite to reduce the samples needed to cover the data values for variables in a program. Ball [Bal04] proposes predicate-complete coverage with the goal of covering all reachable observable states defined by program predicates (either specified by programmers or generated through automatic predication abstractions [GS97, VPP00, BMMR01]). These program predicates also partition the data values for variables in a program.

Recently a specification-based test adequacy criterion called *bounded exhaustive testing* [MK01, BKM02, SYC+04, Khu03, Mar05] has been proposed to test a program, especially one that has structurally complex inputs. Bounded exhaustive testing tests a program on all *valid* inputs up to a given bound; the numeric bound, called the scope, is defined for the size of input structures. Experiments [MAD+03, Khu03, SYC+04, Mar05] have shown that exhaustive testing within a small bound can produce a high-quality test suite in terms of fault detection capability and code coverage. Test generation techniques for bounded exhaustive testing often require developers to specify a class

invariant [LBR98, LG00], which describes the properties of a valid input structure, and a range of (sampled) data values for non-reference-type variables in an input structure. In bounded exhaustive testing, developers can specify a scope so that testing stops when a program is tested on all valid inputs up to the scope. Alternatively, without requiring a predefined scope, exhaustive testing can test a program on all valid inputs by starting from the smallest ones and iteratively increasing the input size until time runs out. Our research on test generation is a type of bounded exhaustive testing but does not require specifications.

2.2 Test Generation

Generating test inputs for an object-oriented program involves two tasks: (1) directly constructing relevant receiver-object states or indirectly constructing them through method sequences, and (2) generating relevant method arguments. For the first task, some specification-based approaches rely on a user-defined class invariant [LBR98, LG00] to know whether a directly-constructed receiver-object state is valid, and to directly construct all valid receiver-object states up to a given bound. TestEra [MK01, Khu03] relies on a class invariant written in the Alloy language [JSS01] and systematically generates tests by using Alloy Analyzer [JSS00], which does bounded-exhaustive, SAT-based checking. Korat [BKM02, Mar05] relies on an imperative predicate, an implementation for checking class invariants. Korat monitors field accesses within the execution of an imperative predicate and uses this information to prune the search for all valid object states up to a given bound. Inspired by Korat, the AsmLT model-based testing tool [GGSV02, Fou] also includes a solver for generating bounded-exhaustive inputs based on imperative predicates. Some other test generation approaches rely on an application-specific state generator to construct valid receiver-object states. Ball et al. [BHR+00] present a combinatorial algorithm for generating states based on a dedicated generator for complex data structures. Different from these previous approaches, our test generation approach does not require class invariants or dedicated state generators because our approach does not directly construct receiver-object states but indirectly constructs receiver-object states through bounded-exhaustive method sequences.

Some test generation tools also generate different method sequences for an object-oriented program. Tools based on (smart) random generation include Jtest [Par03] (a commercial tool for

Java) and Eclat [PE05] (a research prototype for Java). Tonella [Ton04] uses a genetic algorithm to evolve a randomly generated method sequence in order to achieve higher branch coverage. Buy et al. [BOP00] use data flow analysis, symbolic execution, and automated deduction to generate method sequences exercising definition-use pairs of object fields. Our test generation approach generates bounded-exhaustive tests, which can achieve both high code coverage and good fault-detection capability, whereas these previous approaches cannot guarantee the bounded-exhaustiveness of the generated tests. Like our approach, both Java Pathfinder input generator [VPK04] and the AsmLT model-based testing tool [GGSV02, Fou] use state exploration techniques [CGP99] to generate bounded-exhaustive method sequences but these two tools require developers to carefully choose sufficiently large concrete domains for method arguments and AsmLT additionally requires developers to choose the right abstraction functions to guarantee the bounded-exhaustiveness. Our approach uses symbolic execution to automatically derive relevant arguments and explore the symbolic-state space, whose size is much smaller than the concrete-state space explored by Java Pathfinder input generator and AsmLT.

Existing test generation tools use different techniques to achieve the second task in object-oriented test generation: generating relevant method arguments. Both TestEra [MK01, Khu03] and Korat [BKM02, Mar05] use a range of user-defined values for generating primitive-type arguments (as well as primitive-type fields in receiver-object states) and use their bounded-exhaustive testing techniques to generate reference-type arguments if their class invariants are provided. In order to generate primitive-type arguments, some tools such as JCrasher [CS04] and Eclat [PE05] use predefined default values or random values for specific primitive types. For a non-primitive-type argument, these tools use random method sequences where the last method call's return is of the non-primitive type. Jtest [Par03] uses symbolic execution [Kin76, Cla76] and constraint solving to generate arguments of primitive types. Java Pathfinder input generator [KPV03, VPK04] can generate both method arguments and receiver-object states by using symbolic execution and constraint solving; its test generation feature is implemented upon its explicit-state model checker [VHBP00].

Symbolic execution is also the foundation of static code analysis tools. These tools typically do not generate test data, but automatically verify simple properties of programs. Recently, tools such as SLAM [BMMR01, Bal04] and Blast [HJMS03, BCM04] were adapted for generating inputs to test C programs. However, neither of them can deal with complex data structures, which are the

focus of our test generation approach. Our test generation approach also uses symbolic execution; however, in contrast to the existing testing tools that use symbolic execution, our test generation approach uses symbolic execution to achieve both tasks (generation of receiver-object states and method arguments) systematically without requiring class invariants.

2.3 Test Selection

There are different definitions of test selection. One definition of test selection is related to test generation (discussed in Section 2.2): selecting which test inputs to generate. Some other definitions of test selection focus on selecting tests among tests that have been generated because it is costly to run, rerun, inspect, or maintain all the generated tests. Our test selection approach focuses on selecting tests for inspection.

Test adequacy criteria (discussed in Section 2.1) can be directly used to guide test selection: a test is selected if the test can enhance the existing test suite toward satisfying a test adequacy criterion. In partition testing [Mye79], a test input domain is divided into subdomains based on some criteria (such as those test adequacy criteria discussed in Section 2.1), and then we can select one or more representative tests from each subdomain. If a subdomain is not covered by the existing test suite, we can select a generated test from that subdomain.

Pavlopoulou and Young [PY99] proposed residual structural coverage to describe the structural coverage that has not been achieved by the existing test suite. If the execution of a later generated test exercises residual structural coverage, the test is selected for inspection and inclusion in the existing test suite. If we use residual statement coverage or branch coverage in test selection, we may select only a few tests among generated tests although many unselected tests may provide new value like exposing new faults or increasing our confidence on the program. But if we use residual path coverage, we may select too many tests among generated tests although only some of the selected tests may provide new value. Instead of selecting every test that covers new paths, Dickinson et al. [DLP01a, DLP01b] use clustering analysis to partition executions based on path profiles, and use sampling techniques to select executions from clusters for observations. Regression test prioritization techniques [WHLB97, RUC01, ST02], such as additional structural coverage techniques, can produce a list of sorted tests for regression testing; the same idea can also be applied

to prioritize tests for inspection. Our test selection approach complements these existing structural-coverage-based test selection approaches because our approach operates in the semantic domain of program properties rather than the syntactic domain of program text, which is used by previous program-based test selection approaches.

Goodenough and Gerhart [GG75] discussed the importance of specification-based test selection in detecting errors of omission. Chang and Richardson use specification-coverage criteria for selecting tests that exercise new aspects of a unit's specifications [CR99]. Given algebraic specifications [GH78] a priori, several testing tools [GMH81, BGM91, DF94, HS96, CTCC98] generate and select a set of tests to exercise these specifications. Unlike these specification-based approaches, our test selection approach does not require specifications a priori but uses Daikon [ECGN01] to infer operational abstractions, which are used to guide test selection.

Harder et al. [HME03] present a testing technique based on operational abstractions [ECGN01]. Their operational difference technique starts with an operational abstraction generated from an existing test suite. Then it generates a new operational abstraction from the test suite augmented by a candidate test case. If the new operational abstraction differs from the previous one, it adds the candidate test case to the suite. This process is repeated until some number n of candidate cases have been consecutively considered and rejected. Both the operational difference approach and our approach use the operational abstractions generated from test executions. Our approach exploits operational abstractions' guidance to test generation, whereas the operational difference approach operates on a fixed set of given tests. In addition, their operational difference approach selects tests mainly for regression testing, whereas our approach selects tests mainly for inspection.

Hangal and Lam's DIDUCE tool [HL02] detects bugs and tracks down their root causes. The DIDUCE tool can continuously check a program's behavior against the incrementally inferred operational abstractions during the run(s), and produce a report of all operational violations detected along the way. A usage model of DIDUCE is proposed, which is similar to the unit-test selection problem addressed by our test selection approach. Both DIDUCE and our approach make use of violations of the inferred operational abstractions. The inferred abstractions used by our approach are produced by Daikon [ECGN01] at method entry and exit points, whereas DIDUCE infers a limited set of simpler abstractions from procedure call sites and object/static variable access sites. Also DIDUCE does not investigate the effects of operational abstractions on test generation.

Our redundant-test detection can be seen as a type of test selection: selecting non-redundant tests out of automatically generated tests. Our test selection approach minimizes generated tests by selecting a small number of most useful tests for inspection, whereas our redundant-test detection approach tries to conservatively minimize generated tests from the other end: removing useless tests. Our redundant-test detection detects no redundant tests among tests generated by some tools, such as TestEra [MK01], Korat [BKM02], and Java Pathfinder input generator [VPK04], because these tools intentionally avoid generating redundant tests in their test generation process. Different from the redundant-test avoidance mechanisms built in these tools, the mechanisms in our redundant-test detection are more general and can be embedded in any test generation tools as a part of the test generation process or a post-processing step after the test generation process.

2.4 Regression Testing

Regression testing validates a modified program by retesting it. Regression testing is used to ensure that no new errors are introduced to a previously tested program when the program is modified. Because it is often expensive to rerun all tests after program modifications, one major research effort in regression testing is to reduce the cost of regression testing without sacrificing the benefit or sacrificing as little benefit as possible. For example, when some parts of a program are changed, regression test selection techniques [CRV94, RH97, GHK+01] select a subset of the existing tests to retest the new version of the program. A *safe* regression test selection technique [RH97] ensures that the selected subset of tests contain all the tests that execute the code that was modified from the old version to the new version. Sometimes the available resource might not even allow rerunning the subset of regression tests selected by regression test selection techniques. Recently regression test prioritization techniques [WHLB97, RUCH01, EMR02] have been proposed to order regression tests such that their execution provides benefits such as earlier detection of faults.

Regression-test quality is not always sufficient in exhibiting output differences caused by newly introduced errors in a program. Some previous test-generation approaches generate new tests to exhibit behavior deviations caused by program changes. For example, DeMillo and Offutt [DO91] developed a constraint-based approach to generate unit tests that can exhibit program-state deviations caused by the execution of a slightly changed program line (in a mutant produced during

mutation testing [DLS78, How82]). Korel and Al-Yami [KAY98] created driver code that compares the outputs of two program versions, and then leveraged the existing white-box test-generation approaches to generate tests for which the two versions will produce different outputs. However, this type of test-generation problem is rather challenging and it is in fact an undecidable problem. Our regression testing research tries to tackle the problem by exploiting the existing regression tests and checking more-detailed program behavior exercised inside the program.

Regression testing checks whether the behaviors of two program versions are the same given the same test input. Reps et al. [RBDL97] proposed a *program spectrum*[1] to characterize a program's behavior. One of the earliest proposed program spectra are *path spectra* [BL96, RBDL97, HRS+00], which are represented by the executed paths in a program. Harrold et al. [HRS+00] later proposed several other types of program spectra and investigated their potential applications in regression testing. Most of these proposed spectra are defined by using the structural entities exercised by program execution. We refer to these types of program spectra as *syntactic spectra*. Harrold et al. [HRS+00] empirically investigated the relationship between syntactic spectra differences and output differences of two program versions in regression testing. Their experimental results show that when a test input causes program output differences between versions, the test input is likely to cause syntactic spectra differences. However their results show that the reverse is not true. Our regression testing research takes advantage of this phenomenon to expose more behavioral deviations by comparing program spectra instead of just comparing program outputs in regression testing. To better characterize program behavior in regression testing, our research proposes a new class of program spectra, value spectra, that enriches the existing program spectra family. Value spectra are defined by using program states (variable values) and we refer to this type of program spectra as *semantic spectra*. Ernst [Ern00, ECGN01] developed the Daikon tool to infer operational abstractions from program execution and these dynamically inferred abstractions can also be considered as a type of semantic spectra.

Memon et al. [MBN03] model a GUI state in terms of the widgets that the GUI contains, their properties, and the values of the properties. A GUI state corresponds to a function-entry or function-exit state in our approach. Their experimental results show that comparing more-detailed GUI

[1] The name of *spectrum* comes from *path spectrum* [BL96, RBDL97], which is a distribution of paths derived from a run of the program.

states (e.g., GUI states associated with all or visible windows) from two versions can detect faults more effectively than comparing less-detailed GUI states (e.g., GUI states associated with the active window or widget). Our approach also shows that checking more-detailed behavior inside the black box can more effectively expose behavioral deviations than checking just the black-box output. Our approach differs from their approach in two main aspects: our approach is not limited to GUI applications and our approach additionally investigates deviation propagation and deviation-root localization.

Abramson et al. [AFMS96] developed the relative debugging technique that uses a series of user-defined assertions between a reference program and a suspect program. These assertions specify key data structures that must be equivalent at specific locations in two programs. Then a relative debugger automatically compares the data structures and reports any differences while both versions are executed concurrently. Our approach does not require user-defined assertions but compares states at the entries and exits of user functions. The relative debugging technique mainly aims at those data-centric scientific programs that are ported to, or rewritten for, another computer platform, e.g., a sequential language program being ported to a parallel language. Our approach can be applied in the evolution of a broader scope of programs.

Jaramillo et al. [JGS02] developed the comparison checking approach to compare the outputs and values computed by source level statements in the unoptimized and optimized versions of a source program. Their approach requires the optimizer writer to specify the mappings between the unoptimized and optimized versions in the optimization implementation. Their approach locates the earliest point where the unoptimized and optimized programs differ during the comparison checking. Our approach operates at the granularity of user-function executions and uses two heuristics to locate deviation roots instead of using the earliest deviation points. Moreover, our approach does not require any extra user inputs and targets at testing general applications rather than optimizers in particular.

2.5 Behavior Inference

Ernst et al. [ECGN01] developed the Daikon tool to dynamically infer operational abstractions from test executions. Operational abstractions are reported in the form of axiomatic specifica-

tions [Hoa69, Gri87]. Our test selection approach uses these operational abstractions to guide test generation and selection. These abstractions describe the observed relationships among the values of object fields, arguments, and returns of a single method in a class interface, whereas the observer abstractions inferred in our test abstraction approach describe the observed state-transition relationships among multiple methods in a class interface and use the return values of observers to represent object states, without explicitly referring to object fields. Henkel and Diwan [HD03] discover algebraic abstractions (in the form of algebraic specifications [GH78]) from the execution of automatically generated unit tests. Their discovered algebraic abstractions usually present a local view of relationships between two methods, whereas observer abstractions present a global view of relationships among multiple methods. Observer abstractions are a useful form of behavior inference, complementing operational or algebraic abstractions.

Whaley et al. [WML02] extract Java component interfaces from system-test executions. The extracted interfaces are in the form of multiple finite state machines, each of which contains the methods that modify or read the same object field. The observer abstractions inferred by our test abstraction approach are also in the form of multiple finite state machines, each of which is with respect to a set of observers (containing one observer by default). Their approach maps all concrete states that are at the same state-modifying method's exits to the same abstract state. Our test abstraction approach maps all concrete states on which observers' return values are the same to the same abstract state. Although their approach is applicable to system-test executions, it is not applicable to the executions of automatically generated unit tests, because their resulting finite state machine would be a complete graph of methods that modify the same object field. Ammons et al. [ABL02] mine protocol specifications in the form of a finite state machine from system-test executions. Yang and Evans [YE04] also infer temporal properties in the form of the strictest pattern any two methods can have in execution traces. These two approaches face the same problem as Whaley et al.'s approach when being applied on the executions of automatically generated unit tests. In summary, the general approach developed by Whaley et al. [WML02], Ammons et al. [ABL02], or Yang and Evans [YE04] does not capture object states as accurately as our approach and none of them can be applied to the executions of automatically generated unit tests.

Given a set of predicates, predicate abstraction [GS97, BMMR01] maps a concrete state to an abstract state that is defined by the boolean values of these predicates on the concrete state. Given a

set of observers, observer abstraction maps a concrete state to an abstract state that is defined by the return values (not limited to boolean values) of these observers on the concrete state. Concrete states considered by predicate abstractions are usually those program states between program statements, whereas concrete states considered by observer abstractions are those object states between method calls. Predicate abstraction is mainly used in software model checking, whereas observer abstraction in our approach is mainly used in helping inspection of test executions.

Kung et al. [KSGH94] statically extract object state models from class source code and use them to guide test generation. An object state model is in the form of a finite state machine: the states are defined by value intervals over object fields, which are derived from path conditions of method source; the transitions are derived by symbolically executing methods. Our approach dynamically extracts finite state machines based on observers during test executions.

Grieskamp et al. [GGSV02] generate finite state machines from executable abstract state machines. Manually specified predicates are used to group states in abstract state machines to hyper-states during the execution of abstract state machine. Finite state machines, abstract state machines, and manually specified predicates in their approach correspond to observer abstractions, concrete object state machines, and observers in our approach, respectively. However, our approach is totally automatic and does not require programmers to specify any specifications or predicates.

2.6 Feedback Loop in Program Analysis

There have been several lines of static analysis research that use feedback loops to get better program abstractions and verification results. Ball and Rajamani construct a feedback loop between program abstraction and model checking to validate user-specified temporal safety properties of interfaces [BMMR01]. Flanagan and Leino use a feedback loop between annotation guessing and theorem proving to infer specifications statically [FL01]. Guesses of annotations are automatically generated based on heuristics before the first iteration. Human interventions are needed to insert manual annotations in subsequent iterations. Giannakopoulou et al. construct a feedback loop between assumption generation and model checking to infer assumptions for a user-specified property in compositional verification [CGP03, GPB02]. Given crude program abstractions or properties, these feedback loops in static analysis use model checkers or theorem provers to find counterexam-

ples or refutations. Then these counterexamples or refutations are used to refine the abstractions or properties iteratively. Our work is to construct a feedback loop in dynamic analysis, corresponding to the ones in static analysis. Our work does not require users to specify properties, which are inferred from test executions instead.

Naumovich and Frankl propose to construct a feedback loop between finite state verification and testing to dynamically confirm statically detected faults [NF00]. When a finite state verifier detects a property violation, a testing tool uses the violation to guide test data selection, execution, and checking. The tool hopes to find test data that shows the violation to be real. Based on the test information, human intervention is used to refine the model and restart the verifier. This is an example of a feedback loop between static analysis and dynamic analysis. Another example of a feedback loop between static analysis and dynamic analysis is profile-guided optimization [PH90]. Our work focuses instead on the feedback loop on dynamic analysis.

Peled et al. present the black box checking [PVY99] and the adaptive model checking approach [GPY02]. Black box checking tests whether an implementation with unknown structure or model satisfies certain given properties. Adaptive model checking performs model checking in the presence of an inaccurate model. In these approaches, a feedback loop is constructed between model learning and model checking, which is similar to the preceding feedback loops in static analysis. Model checking is performed on the learned model against some given properties. When a counterexample is found for a given property, the counterexample is compared with the actual system. If the counterexample is confirmed, a fault is reported. If the counterexample is refuted, it is fed to the model learning algorithm to improve the learned model. Another feedback loop is constructed between model learning and conformance testing. If no counterexample is found for the given property, conformance testing is conducted to test whether the learned model and the system conform. If they do not conform, the discrepancy-exposing test sequence is fed to the model learning algorithm, in order to improve the approximate model. Then the improved model is used to perform model checking in the subsequent iteration. The dynamic specification inference in our feedback loop is corresponding to the model learning in their feedback loop, and the specification-based test generation in our feedback loop is corresponding to the conformance testing in their feedback loop. Our feedback loop does not require some given properties, but their feedback loop requires user-specified properties in order to perform model checking.

2.7 Conclusion

This chapter has laid out the background for the research developed in this dissertation and discussed how our research is related to other previous research in software testing. In particular, our research does not require specifications; therefore, it is related to program-based or interface-based test adequacy criteria. However, our research operates on the semantic domain of program properties rather than the syntactic domain of program text, which is often the focus of program-based criteria. From test executions, our research infers behavior, which is often in the form of specifications, and further uses the inferred behavior to aid testing activities. In this perspective, our research is also related to specification-based testing. Our test generation approach is a type of bounded-exhaustive testing; however, unlike previous research on bounded-exhaustive testing, our research does not require specifications such as class invariants. Our test generation approach exploits symbolic execution to achieve the generation of both receiver-object states (through method sequences) and relevant method arguments; previous testing research based on symbolic execution either requires specifications or generates relevant arguments for a single method given a specific receiver object. Different from previous testing approaches based on structural coverage, either our redundant-test detection or test selection approach keeps or selects a test if the test exercises new behavior inferred in the semantic domain of program properties; in addition, the inferred behavior is used to guide test generation. Different from previous regression testing approach, which compares the black-box outputs between program versions, our regression testing approach compares the semantic spectra inside the black box. Finally, we have proposed a feedback loop between test generation and behavior inference by using behavior inferred from generated tests to guide better test generation and then using new generated tests to achieve better behavior inference.

Chapter 3

REDUNDANT-TEST DETECTION

Automatic test-generation tools for object-oriented programs, such as Jtest [Par03] (a commercial tool for Java) and JCrasher [CS04] (a research prototype for Java), test a class by generating a test suite for it. A test suite comprises a set of tests, each of which is a sequence of method invocations. When the sequences of two tests are different, these tools conservatively judge that these two tests are not equivalent and thus both are needed. However, there are many situations where different method sequences exercise the same behavior of the class under test. Two sequences can produce *equivalent states* of objects because some invocations do not modify state or because different state modifications produce the same state. Intuitively, invoking the same methods with the same inputs (i.e., the equivalent states of receiver objects and arguments) is redundant. A test is *redundant* if the test includes no new method invocation (i.e., method invocation whose input is different from the input of any method invocation in previous tests). These redundant tests increase the cost of generating, running, inspecting, maintaining a test suite but do not increase a test suite's ability to detect faults or increase developers' confidence on the code under test.

This chapter presents our Rostra approach for detecting redundant tests based on state equivalence. In the Rostra approach, we include five techniques for representing the incoming program state of a method invocation. These five state-representation techniques fall into two types: one is based on the method sequence that leads to the state, and the other is based on concrete states of the objects in the program state. If the representations of two states are the same, we then determine that two states are equivalent. Based on state equivalence, we have defined redundant tests and implemented a tool that dynamically detects redundant tests in an existing test suite. We have evaluated Rostra on 11 subjects taken from a variety of sources. The experimental results show that around 90% of the tests generated by Jtest for all subjects and 50% of the tests generated by JCrasher for almost half of the subjects are redundant. The results also show that removing these redundant tests does not decrease the branch coverage, exception coverage, and fault detection capability of the test

suites.

The next section introduces a running example that is used to illustrate our approach. Section 3.2 presents the five techniques for representing states. Section 3.3 defines state equivalence based on comparing state representation. Section 3.4 defines redundant tests based on state equivalence. Section 3.5 describes the experiments that we conducted to assess the approach and then Section 3.6 concludes.

3.1 Example

We use an integer stack implementation (earlier used by Henkel and Diwan [HD03]) as a running example to illustrate our redundant-test detection techniques. Figure 3.1 shows the relevant parts of the code. The array `store` contains the elements of the stack, and `size` is the number of the elements and the index of the first free location in the stack. The method `push`/`pop` appropriately increases/decreases the size after/before writing/reading the element. Additionally, `push`/`pop` grows/shrinks the array when the `size` is equal to the whole/half of the array length. The method `isEmpty` is an observer that checks if the stack has any elements, and the method `equals` compares two stacks for equality.

The following is an example test suite (written in the JUnit framework [GB03]) with three tests for the `IntStack` class:

```
public class IntStackTest extends TestCase {
  public void test1() {
      IntStack s1 = new IntStack();
      s1.isEmpty();
      s1.push(3);
      s1.push(2);
      s1.pop();
      s1.push(5);
  }

  public void test2() {
      IntStack s2 = new IntStack();
      s2.push(3);
      s2.push(5);
  }
```

```
public class IntStack {
  private int[] store;
  private int size;
  private static final int INITIAL_CAPACITY = 10;
  public IntStack() {
    this.store = new int[INITIAL_CAPACITY];
    this.size = 0;
  }
  public void push(int value) {
    if (this.size == this.store.length) {
      int[] store = new int[this.store.length * 2];
      System.arraycopy(this.store, 0, store, 0, this.size);
      this.store = store;
    }
    this.store[this.size++] = value;
  }
  public int pop() {
    return this.store[--this.size];
  }
  public boolean isEmpty() {
    return (this.size == 0);
  }
  public boolean equals(Object other) {
    if (!(other instanceof IntStack)) return false;
    IntStack s = (IntStack)other;
    if (this.size != s.size) return false;
    for (int i = 0; i < this.size; i++)
      if (this.store[i] != s.store[i]) return false;
    return true;
  }
}
```

Figure 3.1: An integer stack implementation

```
  public void test3() {
      IntStack s3 = new IntStack();
      s3.push(3);
      s3.push(2);
      s3.pop();
  }
}
```

Table 3.1: State representation and comparison

type	technique	representation	comparison
method sequences	WholeSeq	the entire method sequence	equality
	ModifyingSeq	a part of the method sequence	equality
concrete states	WholeState	the entire concrete state	isomorphism
	MonitorEquals	a part of the concrete state	isomorphism
	PairwiseEquals	the entire concrete state	`equals`

A *test suite* consists of a set of tests, each of which is written as a public method. Each *test* has a sequence of method invocations on the objects of the class as well as the argument objects of the class's methods. For example, `test2` creates a stack `s2` and invokes two `push` methods on it. Some existing test-generation tools such as Jtest [Par03] and JCrasher [CS04] generate tests in such a form as specified by the JUnit framework [GB03]. For these generated tests, the correctness checking often relies on the code's design-by-contract annotations [Mey92, LBR98], which are translated into run-time checking assertions [Par03, CL02]. If there are no annotations in the code, the tools only check the robustness of the code: whether the test execution on the code throws uncaught exceptions [CS04].

3.2 State Representation

To define a redundant test (described in Section 3.4), we need to characterize a method invocation's incoming program state, which is called *method-entry state*. A method-entry state describes the receiver object and arguments before a method invocation. Table 3.1 shows the techniques that we use to represent and compare states. Different techniques use different representations for method-entry states and different comparisons of state representations. Each of these five techniques uses one of the two types of information in representing states: 1) method sequences and 2) concrete states of the objects. We next explain the details of these two types and all five techniques.

3.2.1 Method Sequences

Each execution of a test creates several objects and invokes methods on these objects. The representation based on method sequences represents states using sequences of method invocations, following Henkel and Diwan's representation [HD03]. The state representation uses symbolic expressions with the concrete grammar shown below:

exp ::= prim | invoc "`.state`" | invoc "`.retval`"
args ::= ϵ | exp | args "`,`" exp
invoc ::= method "`(`" args "`)`"
prim ::= "`null`" | "`true`" | "`false`" | "`0`" | "`1`" | "`-1`" | ...

Each object or value is represented with an expression. Arguments for a method invocation are represented as sequences of zero or more expressions (separated by commas); the receiver of a non-static, non-constructor method invocation is treated as the first method argument. A static method invocation or constructor invocation does not have a receiver. The `.state` and `.retval` expressions denote the state of the receiver after the invocation and the return of the invocation, respectively. For brevity, the grammar shown above does not specify types, but the expressions are well-typed according to the Java typing rules [AGH00]. A method is represented uniquely by its defining class, name, and the entire signature. For brevity, we do not show a method's defining class or signature in the state-representation examples below.

For example, the state of `s2` at the end of `test2` is represented as

`push(push(<init>().state, 3).state, 5).state`,

where `<init>` represents the constructor that takes no receiver and `<init>().state` represents the object created by the constructor invocation. This object becomes the receiver of the method invocation `push(3)`, and so on.

A method-entry state is represented by using tuples of expressions (two tuples are equivalent if and only if their expressions are component-wise identical). For example, the method-entry state of the last method invocation of `test2` is represented by `<push(<init>().state, 3).state, 5>`, where the first expression `push(<init>().state, 3).state` denotes the receiver-object state and the second expression `5` denotes the argument value. When collecting method sequences for state representation, if a later-encountered expression (or sub-expression) is aliased with an

earlier-encountered expression (or sub-expression) in a method-entry state's representation, we can replace the representation of the later-encountered expression with the identifier of the first-encountered aliased expression in the representation. Under this situation, each non-primitive-type expression in the representation needs to be associated with a unique identifier. For example, consider the following two tests `test4` and `test5`:

```
public void test4() {
  IntStack s4 = new IntStack();
  IntStack s = new IntStack();
  s4.equals(s);
}

public void test5() {
  IntStack s5 = new IntStack();
  s5.equals(s5);
}
```

If we do not consider aliasing relationships among expressions in state representation, the method-entry states of the last method invocation (`equals`) of the both tests are represented by the same expression: `<<init>().state, <init>().state>`. However, these two `equals` method invocations may exhibit different program behaviors if object identities are compared during the `equals` method executions. After aliasing relationships are considered, the method-entry state representation of `equals` in `test4` is different from the one in `test5`, which is then represented by `<<init>().state@1, @1>`, where `@1` denotes the identifier of `t5`.

The state representation based on method sequences allows tests to contain loops, arithmetic, aliasing, and/or polymorphism. Consider the following manually written tests `test6` and `test7`:

```
public void test6() {
  IntStack t = new IntStack();
  IntStack s6 = t;
  for (int i = 0; i <= 1; i++)
     s6.push(i);
}

public void test7() {
  IntStack s7 = new IntStack();
  int i = 0;
```

```
    s7.push(i);
    s7.push(i + 1);
  }
```

Our implementation dynamically monitors the invocations of the methods on the actual objects created in the tests and collects the actual argument values for these invocations.[1] It represents each object using a method sequence; for example, it represents both `s6` and `s7` at the end of `test6` and `test7` as `push(push(<init>().state, 0).state, 1).state`.

We next describe two techniques that include different methods in the method sequences for state representation: WholeSeq and ModifyingSeq.

WholeSeq

This WholeSeq technique represents the state of an object with an expression that includes *all* methods invoked on the object since it has been created, including the constructor. Our implementation obtains this representation by executing the tests and keeping a mapping from objects to their corresponding expressions.

Recall that each method-entry state is represented as a tuple of expressions that represent the receiver object and the arguments. Two state representations are equivalent if and only if the tuples are identical. For example, WholeSeq represents the states before `push(2)` in `test3` and `test1` as `<push(<init>().state, 3).state, 2>` and `<push(isEmpty(<init>().state).state, 3).state, 2>`, respectively, and these two state representations are not equivalent.

The WholeSeq technique maintains a table that maps each object to a method sequence that represents that object. At the end of each method call, the sequence that represents the receiver object is extended with the appropriate information that represents the call.

ModifyingSeq

The ModifyingSeq technique represents the state of an object with an expression that includes *only* those methods that modified the state of the object since it has been created, including the construc-

[1] Although our implementation needs to run the tests to detect redundant tests and the cost of running redundant tests is not saved, Section 3.4 presents the practical applications of our approach.

tor. Our implementation monitors the method executions to determine at run time whether they modify the state.

Similar to the WholeSeq technique, the ModifyingSeq technique also maintains a table that maps each object to a method sequence that represents that object. The sequence is extended with the appropriate information that represents the call only when the method execution has modified the receiver. ModifyingSeq dynamically monitors the execution and determines that the receiver is modified if there is a write to a field that is reachable from the receiver. ModifyingSeq builds and compares method-entry states in the same way as WholeSeq; however, because ModifyingSeq uses a coarser representation for objects than WholeSeq, ModifyingSeq can find the representations of more method-entry states to be equivalent. For example, `isEmpty` does not modify the state of the stack, so ModifyingSeq represents the states before `push(2)` in both `test3` and `test1` as `<push(<init>().state, 3).state, 2>` and thus finds their representations to be equivalent.

3.2.2 Concrete States

The execution of a method operates on the program state that includes a program heap. The representation based on concrete states considers only parts of the heap that are relevant for affecting a method's execution; we also call each part a "heap" and view it as a graph: nodes represent objects and edges represent fields. Let P be the set consisting of all primitive values, including `null`, integers, etc. Let O be a set of objects whose fields form a set F. (Each object has a field that represents its class, and array elements are considered index-labeled object fields.)

Definition 1. *A* heap *is an edge-labelled graph* $\langle O, E\rangle$*, where* $E = \{\langle o, f, o'\rangle | o \in O, f \in F, o' \in O \cup P\}$.

Heap isomorphism is defined as graph isomorphism based on node bijection [BKM02].

Definition 2. *Two heaps* $\langle O_1, E_1\rangle$ *and* $\langle O_2, E_2\rangle$ *are* isomorphic *iff there is a bijection* $\rho : O_1 \to O_2$ *such that:*

$$\begin{aligned} E_2 \;=\; & \{\langle \rho(o), f, \rho(o')\rangle | \langle o, f, o'\rangle \in E_1, o' \in O_1\} \cup \\ & \{\langle \rho(o), f, o'\rangle | \langle o, f, o'\rangle \in E_1, o' \in P\}. \end{aligned}$$

```
Map ids; // maps nodes into their unique ids
int[] linearize(Node root, Heap <O,E>) {
  ids = new Map();
  return lin(root, <O,E>);
}
int[] lin(Node root, Heap <O,E>) {
  if (ids.containsKey(root))
    return singletonSequence(ids.get(root));
  int id = ids.size() + 1;
  ids.put(root, id);
  int[] seq = singletonSequence(id);
  Edge[] fields = sortByField({ <root, f, o> in E });
  foreach (<root, f, o> in fields) {
    if (isPrimitive(o))
      seq.add(uniqueRepresentation(o));
    else
      seq.append(lin(o, <O,E>));
  }
  return seq;
}
```

Figure 3.2: Pseudo-code of linearization

The definition allows only object identities to vary: two isomorphic heaps have the same fields for all objects and the same values for all primitive fields.

Because only parts of the program heap before a method invocation are relevant for affecting the method's execution, a method-entry state of a method invocation is represented with *rooted* heaps, instead of the whole program heap.

Definition 3. *A rooted heap is a pair $\langle r, h \rangle$ of a root object r and a heap h whose all nodes are reachable from r.*

Although no polynomial-time algorithm is known for checking isomorphism of general graphs, it is possible to efficiently check isomorphism of rooted heaps. Our implementation *linearizes* rooted heaps into sequences such that checking heap isomorphism corresponds to checking sequence equality. Figure 3.2 shows the pseudo-code of the linearization algorithm; similar linearization algorithms [VHBP00, RDHI03, Ios02, AQR$^+$04] have been used in model checking [CGP99]. The

linearization algorithm traverses the entire rooted heap in the depth-first order, starting from the root. When the algorithm visits a node for the first time, it assigns a unique identifier to the node, keeping this mapping in `ids` to use again for nodes that appear in cycles. We can show that the linearization normalizes rooted heaps into sequences.

Theorem 1. *Two rooted heaps* $\langle o_1, h_1 \rangle$ *and* $\langle o_2, h_2 \rangle$ *are isomorphic iff* $\texttt{linearize}(o_1, h_1) = \texttt{linearize}(o_2, h_2)$.

We next describe three techniques that use concrete states in state representation: WholeState, MonitorEquals, and PairwiseEquals.

WholeState

The WholeState technique represents the method-entry state of a method invocation using the heap rooted from the receiver object and the arguments.[2] Two state representations are equivalent iff the sequences obtained from their linearized rooted heaps are identical. Our implementation uses Java reflection [AGH00] to recursively collect all the fields that are reachable from the receiver and arguments before a method invocation.

For example, the following left and right columns show the state representations of `s1` and `s2` before `push(5)` in `test1` and `test2`, respectively:

```
// s1 before push(5)      // s2 before push(5)
store = @1                store = @1
store.length = 10         store.length = 10
store[0] = 3              store[0] = 3
store[1] = 2              store[1] = 0
store[2] = 0              store[2] = 0
...                       ...
store[9] = 0              store[9] = 0
size = 1                  size = 1
```

In both state representations, being of the integer array type, the `store` field is considered as a node (not being a primitive value); therefore, the linearization algorithm assigns a unique identifier

[2]The linearization algorithm in Figure 3.2 assumes only one root; however, the method-entry state of a method invocation is represented by the heap rooted from multiple nodes including both the receiver object and the arguments, when some arguments are also object references. To handle multiple roots, we can create a virtual node that points to the receiver object and the arguments, and then use the algorithm to linearize the heap rooted from this virtual node.

`@1` to `store`. These two state representations are not equivalent, because the primitive values of the `store[1]` field are different.

MonitorEquals

Like WholeState, MonitorEquals also represents a state with a rooted heap, but this heap is only a subgraph of the entire rooted heap. The MonitorEquals technique leverages user-defined `equals` methods to extract only the relevant parts of the rooted heap. MonitorEquals obtains the values $\langle v_0, \ldots, v_n \rangle$ of a method invocation's receiver and arguments. It then invokes v_i`.equals(`v_i`)` for each non-primitive v_i and monitors the field accesses that these executions make. Then the linearization algorithm in Figure 3.2 is revised to linearize only nodes (fields) that are accessed during the `equals` executions. The rationale behind MonitorEquals is that these executions access only the relevant object fields that define an abstract state.

MonitorEquals represents each method-entry state as a rooted heap whose edges consist only of the accessed fields and the edges from the root. Two state representations are equivalent iff the sequences obtained from their linearized rooted heaps are identical.

For example, the following left and right columns show the state representations of `s1` and `s2` before `push(5)` in `test1` and `test2`, respectively:

```
// s1.equals(s1)          // s2.equals(s2)
// before s1.push(5)      // before s2.push(5)
store = @1                store = @1
store[0] = 3              store[0] = 3
size = 1                  size = 1
```

The execution of `s1.equals(s1)` or `s2.equals(s2)` before `push(5)` accesses only the fields `size`, `store`, and elements of `store` whose indices are up to the value of `size`. Then although WholeState finds the state representations of the method-entry states before `push(5)` in `test1` and `test2` are not equivalent, MonitorEquals find them to be equivalent.

To collect the representation for the method-entry state of a method invocation, our implementation inserts at the method entry the code that invokes v_i`.equals(`v_i`)` for the receiver and each non-primitive argument v_i before a method invocation. It then inserts code before field-access bytecode instructions to monitor their executions so that it can collect all fields that are accessed within

the `equals` executions. The MonitorEquals technique needs to carefully avoid the common optimization pattern that compares the receiver and the argument for identity `this == other` within `equals` methods; if the pattern appears within `equals` methods, MonitorEquals may collect fewer fields than desired.

PairwiseEquals

Like MonitorEquals, the PairwiseEquals technique also leverages user-defined `equals` methods to consider only the relevant parts of the rooted heap. It implicitly uses the entire program heap to represent method-entry states. However, it does not compare (parts of) states by isomorphism. Instead, it runs the test to build the concrete objects that correspond to the receiver and arguments, and then uses the `equals` method to compare pairs of states. It assigns a unique identifier to a state s_1 as its state representation if there exists no previously encountered state s_2 such that s_1`.equals(`s_2`)` returns `true`; otherwise, s_1's representation is the unique identifier assigned to s_2. The state representations of two states s_1 and s_2 are equivalent iff the states' assigned identifiers are identical (that is, s_1`.equals(`s_2`)` returns `true`).

PairwiseEquals can find more object's representations to be equivalent than MonitorEquals. For example, consider a class that implements a set using an array. PairwiseEquals reports the representations of two objects to be equivalent if they have the same set of array elements, regardless of the order, whereas MonitorEquals reports the representations of two objects with the same elements but different order to be nonequivalent. However, when representing the method-entry state of a method invocation, unlike MonitorEquals, PairwiseEquals fails to include aliasing relationships among the receiver, arguments, and their object fields. For example, the method-entry state representations of `equals` in both `test4` and `test5` are the same, being `<e1, e1>`, where `e1` is the identifier assigned to `s`, `s4`, and `s5`.

Our implementation collects the objects for the receiver and arguments and then compares them by using Java reflection [AGH00] to invoke `equals` methods. Note that subsequent test execution can modify these objects, so PairwiseEquals needs to reproduce them for later comparison. Our implementation re-executes method sequences to reproduce objects; an alternative would be to maintain a copy of the objects.

3.3 State Equivalence

In the previous section (Section 3.2), we have presented five techniques for representing the method-entry state of a method invocation, and have also described how to determine whether two *state representations* are *equivalent*. Our objective is to determine whether two *method-entry states* are *equivalent* such that invoking the same method on these two method-entry states exhibits the same program behavior, thus having the same fault-detection capability. Several previous projects [BGM91, DF94, HD03] defined state equivalence by using observational equivalence [DF94, LG00]. However, checking it precisely is expensive: by definition it takes infinite time (to check all method sequences), so we use state-representation equivalence presented in the previous section to approximate state equivalence. Observational equivalence, as well as our whole approach, assumes that method executions are deterministic. For example, it is assumed that there is no randomness or multi-threading interaction during method executions; otherwise, different executions for the same method input may produce different results, so model-checking techniques [CGP99] may be more applicable than testing.

When we use state-representation equivalence to approximate state equivalence, the five techniques have different tradeoffs in the following aspects:

Safety. We want to keep two method executions if their method invocations are on two nonequivalent method-entry states; otherwise, discarding one of them may decrease a test suite's fault-detection capability. Our approximation is *safe* (or conservative) if the approximation produces no false negative, where a false negative is defined as a state that is not equivalent to another one but their state representations are equivalent.

Precision. We want to reduce the testing efforts spent on invoking methods on equivalent method-entry states; therefore, we want to reduce false positives, where a false positive is defined as a state that is equivalent to another one but their state representations are not equivalent.

Requirements. Different techniques have different requirements in the access of the bytecode under test, time overhead, space overhead, etc.

3.3.1 Safety

We next discuss under what conditions our techniques are not safe and propose extensions for our techniques to make our techniques safe. Two techniques based on method sequences (WholeSeq and ModifyingSeq) are not safe: because the grammar shown in Section 3.2.1 does not capture a method execution's side effect on an argument, a method can modify the state of a non-primitive-type argument and this argument can be used for another later method invocation. Following Henkel and Diwan's suggested extension [HD03], we can enhance the first grammar rule to address this issue:

$$\text{exp ::= prim} \mid \text{invoc ``}\texttt{.state}\text{''} \mid \text{invoc ``}\texttt{.retval}\text{''} \mid \text{invoc ``}.arg_i\text{''}$$

where the added expression (invoc "$.arg_i$") denotes the state of the modified ith argument after the method invocation.

Two techniques based on method sequences (WholeSeq and ModifyingSeq) are not safe if test code modifies directly some public fields of an object without invoking any of its methods, because these side effects on the object are not captured by method sequences. To address this issue, the implementation of the techniques can be extended to create a public field-writing method for each public field of the object, and monitor the static field access in the test code. If our implementation detects at runtime the execution of a field-write instruction in test code, it inserts a corresponding field-writing method invocation in the method-sequence representation.

WholeState, MonitorEquals, and PairwiseEquals are not safe when the execution of a method accesses some public static fields that are not reachable from the receiver or arguments, or accesses the content of a database or file uncontrolled through the receiver or arguments. We can use static analysis to determine a method execution's extra inputs besides the receiver and arguments, and then collect the state of these extra inputs as a part of the method-entry state.

Two techniques based on user-defined `equals` methods (MonitorEquals and PairwiseEquals) are not safe if the `equals` methods are implemented not to respect observation equivalence, such as not respecting the contract in `java.lang.Object` [SM03]. The contract requires that each `equals` implements an equivalence relation, i.e., it should be reflexive, symmetric, and transitive. In practice, we have found most `equals` methods to implement observational equivalence; however, if `equals` is weaker (i.e., returns `true` for some objects that are not observationally equivalent),

our techniques based on `equals` may not be safe. Although the user need to carefully implement the `equals` methods in order to guarantee the safety, our implementation can dynamically check an approximation of observational equivalence for `equals` and help the user tune the method.

PairwiseEquals is not safe when aliasing relationships among the receiver, arguments, and their object fields can affect the observational equivalence, because PairwiseEquals cannot capture aliasing relationships in its representation, as we discussed in Section 3.2.

3.3.2 Precision

When all five techniques are safe, determined by the mechanisms of representing states, their precision is in increasing order from the lowest to highest: WholeSeq, ModifyingSeq, WholeState, MonitorEquals, and PairwiseEquals. We next discuss under what conditions one technique may generally achieve higher precision than its preceding technique in the list. ModifyingSeq may achieve higher precision than WholeSeq when there are invocations of state-preserving methods (e.g., `isEmpty`) and these invocations appear in method sequences that represent object states. WholeState may achieve higher precision than ModifyingSeq when there are invocations of state-modifying methods (e.g., `remove`) that revert an object's state back to an old state that was reached previously with a shorter method sequence. MonitorEquals may achieve higher precision than WholeState when some fields of an object are irrelevant for affecting observational equivalence. PairwiseEquals may achieve higher precision than MonitorEquals when there are two objects s_1 and s_2 where s_1`.equals(`s_2`)` returns `true` but they have different linearized heaps that consist of fields accessed within s_1`.equals(`s_1`)` or s_2`.equals(`s_2`)`. The precision of MonitorEquals or PairwiseEquals relies on the user-defined `equals` method. If `equals` is stronger (i.e., returns `false` for two objects that are observationally equivalent), MonitorEquals or PairwiseEquals may not achieve 100% precision.

3.3.3 Requirements

Our implementations of five techniques operate on Java bytecode without requiring Java source code. Unlike WholeState or MonitorEquals, our implementation of WholeSeq, ModifyingSeq, or PairwiseEquals does not require to access the internal states or the bytecode of the class under test.

These three techniques can be applied when the internal states or the bytecode of the class under test are not available, for example, when testing components [HC01] or web services [ACKM02]. Although our implementation of WholeSeq or ModifyingSeq uses dynamic analysis, we can perform a static analysis on the test code to gather the method sequence without executing the test code. Although this static analysis would be conservative and less precise than the dynamic analysis, it would enable the determination of state equivalence and the detection of redundant tests (described in the next section) without executing them.

Generally WholeSeq and ModifyingSeq require less analysis time than WholeState and MonitorEquals, because WholeSeq and ModifyingSeq do not require the collection of object-field values. ModifyingSeq requires more time than WholeSeq, because our implementation of ModifyingSeq also needs to dynamically determine whether a method execution is a state-modifying one. When there are a relatively large number of nonequivalent states, PairwiseEquals typically requires more time than MonitorEquals because PairwiseEquals compares the state under consideration with those previously encountered nonequivalent objects one by one, whereas our implementation of MonitorEquals uses efficient hashing and storing to check whether the state under consideration is equivalent to one of those previously encountered states, because we know the representation (sequence).

ModifyingSeq requires less space than WholeSeq. When tests contain relatively short sequences, WholeSeq or ModifyingSeq may require less space than WholeState or MonitorEquals for storing the state representation of a single nonequivalent state; however, the total number of nonequivalent states determined by WholeSeq or ModifyingSeq is larger than the total number of nonequivalent states determined by WholeState or MonitorEquals. MonitorEquals requires less space than WholeState. PairwiseEquals may require less space for storing state representations (being just unique identifiers) than WholeState or MonitorEquals, whose state representations consist of sequences linearized from object fields; however, our implementation of PairwiseEquals needs to keep a copy of each nonequivalent object around for later comparison, as was described in Section 3.2.

3.4 Redundant Tests

We next show how equivalent states give rise to equivalent method executions and define redundant tests and test-suite minimization.

Each test execution produces several method executions.

Definition 4. *A* method execution $\langle m, s\rangle$ *is a pair of a method* m *and a method-entry state* s*.*

We denote by $[\![t]\!]$ the sequence of method executions produced by a test t, and we write $\langle m, s\rangle \in [\![t]\!]$ when a method execution $\langle m, s\rangle$ is in the sequence for t. We define equivalent method executions based on equivalent states.

Definition 5. *Two method executions* $\langle m_1, s_1\rangle$ *and* $\langle m_2, s_2\rangle$ *are* equivalent *iff* $m_1 = m_2$ *and* s_1 *and* s_2 *are equivalent.*

We further consider minimal test suites that contain no redundant tests.

Definition 6. *A test* t *is* redundant *for a test suite* S *iff for each method execution of* $[\![t]\!]$*, there exists an equivalent method execution of some test from* S*.*

Definition 7. *A test suite* S *is* minimal *iff there is no* $t \in S$ *that is redundant for* $S \backslash \{t\}$*.*

Minimization of a test suite S' finds a minimal test suite $S \subseteq S'$ that exercises the same set of nonequivalent method executions as S' does.

Definition 8. *A test suite* S minimizes *a test suite* S' *iff* S *is minimal and for each* $t' \in S'$ *and each* $\langle m', s'\rangle \in [\![t']\!]$*, there exist* $t \in S$ *and* $\langle m, s\rangle \in [\![t]\!]$ *such that* $\langle m', s'\rangle$ *and* $\langle m, s\rangle$ *are equivalent.*

Given a test suite S', there can be several test suites $S \subseteq S'$ that minimize S'. Among the test suites that minimize S', we could find a test suite that has the smallest possible number of tests or the smallest possible total number of method executions for the tests. Finding such test suites reduces to optimization problems called “minimum set cover” and “minimum exact cover”, respectively; these problems are known to be NP complete, and in practice approximation algorithms are used [Joh74]. Our implementation runs the tests in a given test suite with its default test-execution order (such as the one controlled by the JUnit framework [GB03]) and then minimizes the test suite by using a greedy algorithm. Running the tests in different orders can cause our implementation to produce

different minimized test suites; however, these different minimized test suites produce the same total number of nonequivalent method executions.

In particular, our implementation collects method-entry states dynamically during test executions. We use the Byte Code Engineering Library [DvZ03] to instrument the bytecodes of the classes under test at the class-loading time. The instrumentation adds the code for collecting state representations at the entry of each method call in a test. For some techniques, it also adds the code for monitoring instance-field reads and writes. Our instrumentation collects the method signature, the receiver-object reference, and the arguments at the beginning of each method call in the test. The receiver of these calls is usually an instance object of the class under test. The instrumentation does not collect the method-entry states for calls that are internal to these objects. Different techniques also collect and maintain additional information. After finishing running the given test suite, our implementation outputs a minimized test suite in the form of a JUnit test class [GB03].

Our redundant-test detection techniques can be used in the following four practical applications: test-suite assessment, test selection, test-suite minimization, and test generation.

Assessment: Our techniques provide a novel quantitative comparison of test suites, especially those generated by automatic test-generation tools. For each test suite, our techniques can find nonequivalent object states, nonequivalent method executions, and non-redundant tests. We can then use their metrics to compare the quality of different test suites.

Selection: Our techniques can be used to select a subset of automatically generated tests to augment an existing (manually or automatically generated) test suite. We feed the existing test suite and the new tests to our techniques, running the existing test suite first. The minimal test suite that our techniques then produce will contain those new tests that are non-redundant with respect to the existing test suite.

Minimization: Our techniques can be used to minimize an automatically generated test suite for correctness inspection and regression executions. Without a priori specifications, automatically generated tests typically do not have test oracles for correctness checking, and the tester needs to manually inspect the correctness of (some) tests. Our techniques help the tester to focus only on the non-redundant tests, or more precisely the nonequivalent method executions. Running redundant tests is inefficient, and our techniques can remove these tests from a regression test suite. However, we need to be careful because changing the code can make a test that is redundant in the old code

to be non-redundant in the new code. If two method sequences in the old code produce equivalent object states, *and* the code changes do not impact these two method sequences [RT01], we can still safely determine that the two sequences in the new code produce equivalent object states. Additionally, we can always safely use our techniques to perform regression test prioritization [RUC01,ST02] instead of test-suite minimization.

Generation: Existing test-generation tools can incorporate our techniques to avoid generating and executing redundant tests. Although our implementations of the techniques are using dynamic analysis, they can determine whether a method execution me is equivalent to some other execution *before* running me; the method-entry state required for determining equivalence is available before the execution. Test-generation tools that execute tests, such as Jtest [Par03] or AsmLT [GGSV02], can easily integrate our techniques. Jtest executes already generated tests and observes their behavior to guide the generation of future tests. Running Jtest is currently expensive—it spends over 10 minutes generating the tests for relatively large classes in our experiments (Section 3.5)—but much of this time is spent on redundant tests. In the next chapter, we will present how our techniques can be incorporated to generate only non-redundant tests.

3.5 Evaluation

This section presents two experiments that assess how well Rostra detects redundant tests: 1) we investigate the benefit of applying Rostra on tests generated by existing tools; and 2) we validate that removing redundant tests identified by Rostra does not decrease the quality of test suites. We have performed the experiments on a Linux machine with a Pentium IV 2.8 GHz processor using Sun's Java 2 SDK 1.4.2 JVM with 512 MB allocated memory.

3.5.1 Experimental Setup

Table 3.2 lists the 11 Java classes that we use in our experiments. The `IntStack` class is our running example. The `UBStack` class is taken from the experimental subjects used by Stotts et al. [SLA02]. The `ShoppingCart` class is a popular example for JUnit [Cla00]. The `BankAccount` class is an example distributed with Jtest [Par03]. The remaining seven classes are data structures used to evaluate Korat [BKM02, MAD+03]. The first four columns show the class name, the number of

Table 3.2: Experimental subjects

class	meths	public meths	ncnb loc	Jtest tests	JCrasher tests
IntStack	5	5	44	94	6
UBStack	11	11	106	1423	14
ShoppingCart	9	8	70	470	31
BankAccount	7	7	34	519	135
BinSearchTree	13	8	246	277	56
BinomialHeap	22	17	535	6205	438
DisjSet	10	7	166	779	64
FibonacciHeap	24	14	468	3743	150
HashMap	27	19	597	5186	47
LinkedList	38	32	398	3028	86
TreeMap	61	25	949	931	1000

methods, the number of public methods, and the number of non-comment, non-blank lines of code for each subject.

We use two third-party test generation tools, Jtest [Par03] and JCrasher [CS04], to automatically generate test inputs for program subjects. Jtest allows the user to set the length of calling sequences between one and three; we set it to three, and Jtest first generates all calling sequences of length one, then those of length two, and finally those of length three. JCrasher automatically constructs method sequences to generate non-primitive arguments and uses default data values for primitive arguments. JCrasher generates tests as calling sequences with the length of one. The last two columns of Table 3.2 show the number of tests generated by Jtest and JCrasher.

Our first experiment uses the five techniques to detect redundant tests among those generated by Jtest and JCrasher. Our second experiment compares the quality of original and minimized test suites using 1) branch coverage, 2) nonequivalent, uncaught-exception count, and 3) fault-detection capability. We adapted Hansel [Han03] to measure branch coverage and nonequivalent, uncaught-exception count. (Two exceptions are equivalent if they have the same throwing location

and type.) To estimate the fault-detection capability, we use two mutation-analysis tools for Java: Jmutation [MKO02] and Ferastrau [MAD+03]. We select the first 300 mutants (i.e., 300 versions each of which is seeded with a bug) produced by Jmutation and configure Ferastrau to produce around 300 mutants for each subject. We estimate the fault-detection capability of a test suite by using the mutant killing ratio of the test suite, which is the number of the killed mutants divided by the total number of mutants. To determine whether a test kills a mutant, we have written specifications and used the JML runtime verifier [CL02] to compare the method-exit states and returns of the original and mutated method executions.

3.5.2 Experimental Results

Figures 3.3 and 3.4 show the results of the first experiment—the percentage of redundant tests generated—for Jtest and JCrasher, respectively. We also measured the percentages of equivalent object states and equivalent method executions; they have similar distributions as the redundant tests. We observe that all techniques except WholeSeq identify around 90% of Jtest-generated tests to be redundant for all subjects and 50% of JCrasher-generated tests to be redundant for five out of 11 subjects. Possible reasons for higher redundancy of Jtest-generated tests include: 1) Jtest generates more tests; and 2) Jtest-generated tests have longer call length.

We observe a significant improvement achieved by ModifyingSeq over WholeSeq in detecting redundant tests. In Figure 3.3, this improvement for `IntStack` is not so large as the one for other subjects, because `IntStack` has only one state-preserving method (`isEmpty`), whereas other subjects have a higher percentage of state-preserving methods in their class interfaces. There are some improvements achieved by the last three techniques based on concrete states over ModifyingSeq. But there is no significant difference in the results for the last three techniques. We hypothesize that our experimental subjects do not have many irrelevant object fields for defining object states and/or the irrelevant object fields do not significantly affect the redundant test detection.

Figures 3.5 and 3.6 show the elapsed real time of running our implementation to detect redundant tests generated by Jtest and JCrasher, respectively. We observe that the elapsed time is reasonable: it ranges from a couple of seconds up to several minutes, determined primarily by the class complexity and the number of generated tests. In Figures 3.5, the elapsed time of MonitorEquals for

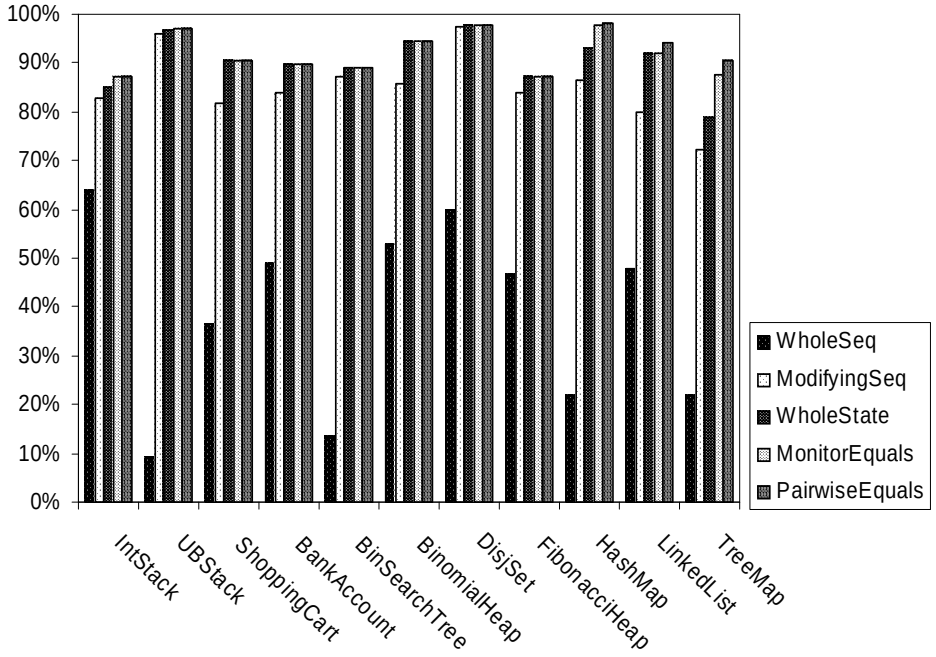

Figure 3.3: Percentage of redundant tests among Jtest-generated tests

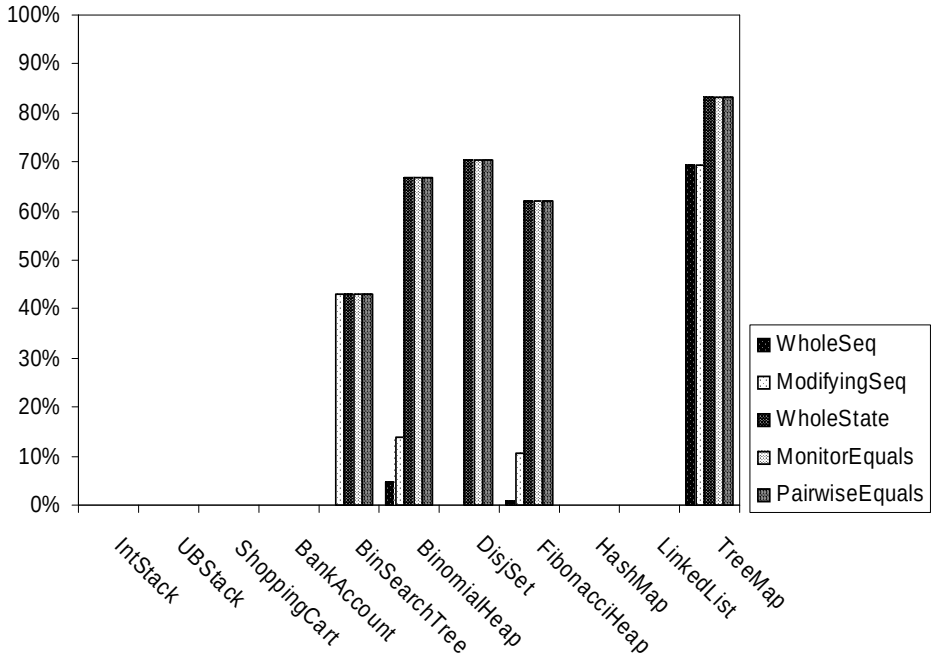

Figure 3.4: Percentage of redundant tests among JCrasher-generated tests

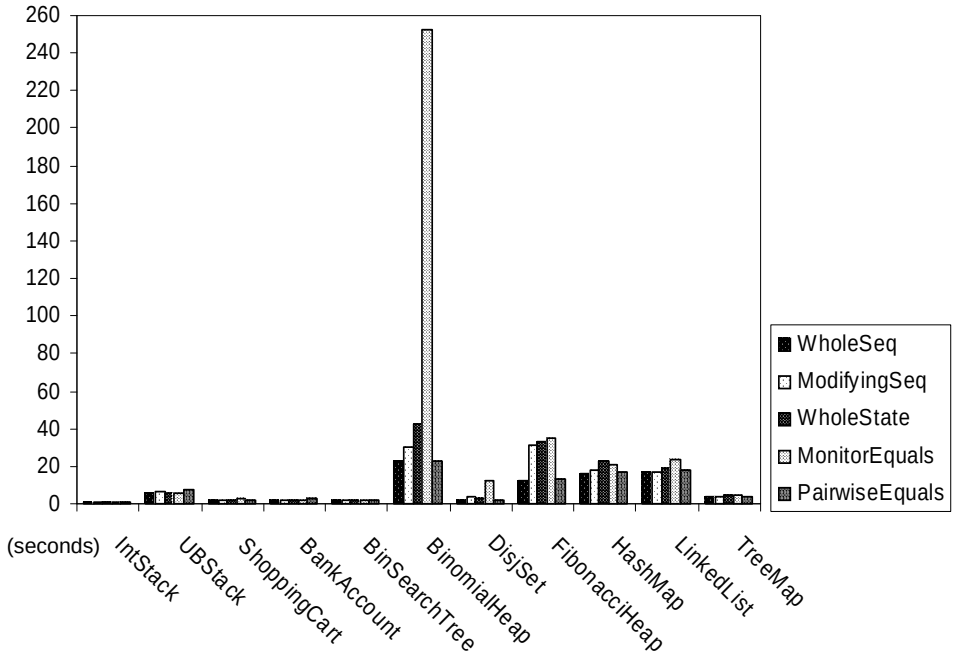

Figure 3.5: Elapsed real time of detecting redundant tests among Jtest-generated tests

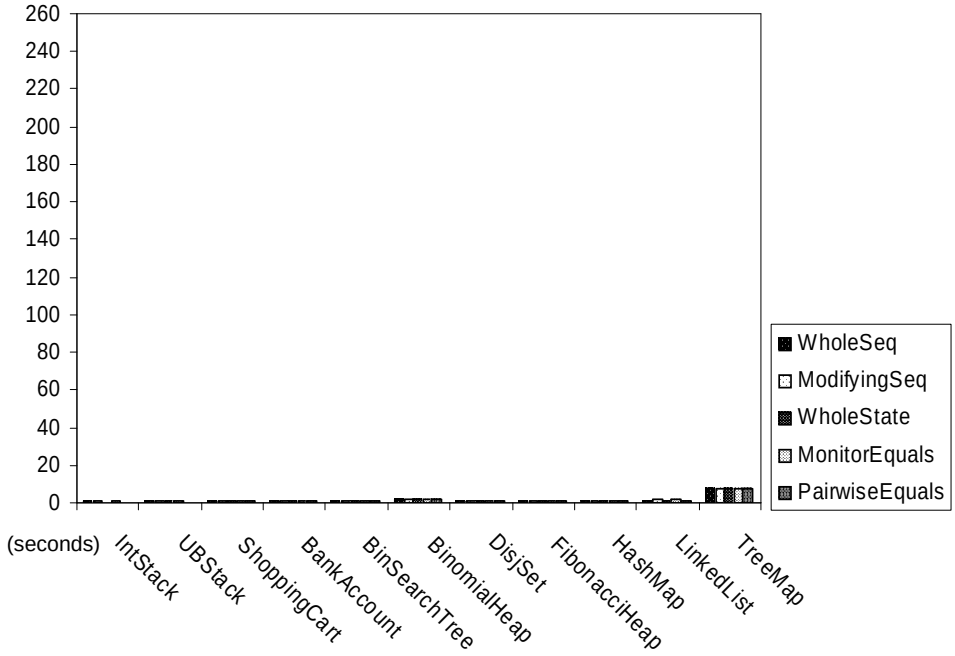

Figure 3.6: Elapsed real time of detecting redundant tests among JCrasher-generated tests

`BinomialHeap` is relatively expensive, because the number of generated tests for `BinomialHeap` is relatively large and invoking its `equals` is relatively expensive.

To put the analysis time of our techniques into perspective, we need to consider the whole test generation: if test-generation tools such as Jtest incorporated our techniques into generation, the time savings achieved by avoiding redundant tests would significantly exceed the extra cost of running our techniques. The next chapter will show how we can avoid generating redundant tests based on our techniques.

Table 3.3 shows the results of the second experiment: nonequivalent, uncaught-exception counts (columns 2 and 3), branch-coverage percentages (columns 4 and 5), killing ratios for Ferastrau mutants (columns 6 and 7), and killing ratios for Jmutation mutants (columns 8 and 9). The columns marked "jte" and "jcr" correspond to Jtest and JCrasher, respectively. The original Jtest-generated and JCrasher-generated test suites have the same measures as their corresponding Rostra-minimized test suites in all cases except for the four cases whose entries are marked with "*". The differences are due only to the MonitorEquals and PairwiseEquals techniques. The minimized Jtest-generated test suites for `IntStack` and `TreeMap` cannot kill three Ferastrau-generated mutants that the original test suites can kill. This shows that minimization based on `equals` can reduce the fault-detection capability of a test suite, but the probability is very low. The minimized Jtest-generated test suites for `HashMap` and `TreeMap` cannot cover two branches that the original test suites can cover. We have reviewed the code and found that two fields of these classes are used for caching; these fields do not affect object equivalence (defined by `equals`) but do affect branch coverage. These four cases suggest a further investigation on the use of `equals` methods in detecting redundant tests as future work.

3.5.3 Threats to Validity

The threats to external validity primarily include the degree to which the subject programs and third-party test generation tools are representative of true practice. Our subjects are from various sources and the Korat data structures have nontrivial size for unit tests. Of the two third-party tools, one—Jtest—is popular and used in industry. These threats could be further reduced by experiments on various types of subjects and third-party tools. The main threats to internal validity include

Table 3.3: Quality of Jtest-generated, JCrasher-generated, and minimized test suites

class	excptn count		branch cov [%]		Ferastrau kill [%]		Jmutation kill [%]	
	jte	jcr	jte	jcr	jte	jcr	jte	jcr
IntStack	1	1	67	50	*45	40	24	23
UBStack	2	0	94	56	57	25	78	37
ShoppingCart	2	1	93	71	57	51	80	20
BankAccount	3	3	100	100	98	98	89	89
BinSearchTree	3	0	67	14	33	5	57	11
BinomialHeap	3	3	90	66	89	34	64	48
DisjSet	0	0	61	51	26	18	40	29
FibonacciHeap	2	2	86	58	73	21	68	35
HashMap	1	1	*72	43	52	23	48	24
LinkedList	19	10	79	48	24	7	25	9
TreeMap	4	3	*33	11	*16	4	16	7

instrumentation effects that can bias our results. Faults in our implementation, Jtest, JCrasher, or other measurement tools might cause such effects. To reduce these threats, we have manually inspected the collected execution traces for several program subjects.

3.6 Conclusion

Object-oriented unit tests consist of sequences of method invocations. Behavior of an invocation depends on the state of the receiver object and method arguments at the beginning of the invocation. Existing tools for automatic generation of object-oriented test suites, such as Jtest and JCrasher for Java, typically ignore this state and thus generate redundant tests that exercise the same method behavior, which increases the testing time without increasing the ability to detect faults.

We have developed five fully automatic techniques for detecting redundant object-oriented unit tests. We have proposed four practical applications of the framework. We have conducted experiments that evaluate the effectiveness of our techniques on detecting redundant tests in test suites

generated by two third-party test-generation tools. The results show that our techniques can substantially reduce the size of these test suites without decreasing their quality. These results strongly suggest that tools and techniques for generation of object-oriented test suites must consider avoiding redundant tests.

Chapter 4

NON-REDUNDANT-TEST GENERATION

Unit tests are becoming an important component of software development. The Extreme Programming discipline [Bec00, Bec03], for example, leverages unit tests to permit continuous and controlled code changes. Although manually created unit tests are valuable, they often do not cover sufficient behavior of the class under test, partly because manual test generation is time consuming and developers often forget to create some important test inputs. While recognizing the importance of unit tests, many companies have provided tools, frameworks, and services around unit tests, ranging from specialized test frameworks, such as JUnit [GB03] or Visual Studio's new team server [Mic04], to automatic unit-test generation tools, such as Parasoft's Jtest [Par03] and Aigtar's Agitator [Agi04]. However, within constrained resources, existing test-generation tools often do not generate sufficient unit tests to fully exercise the behavior of the class under test, for example, by satisfying the branch-coverage test criterion [Bei90], let alone a stronger criterion, such as the bounded intra-method path coverage [BL00] of the class under test. As we have discussed in Chapeter 3, wasting time on generating and running redundant tests is one main reason for existing tools not to generate sufficient unit tests given constrained resources.

In order not to be redundant, a test needs to exercise at least one new method execution (one that is not equivalent to any of those exercised by earlier executed tests). Assume that we have a fixed set of values for method arguments, then in order to generate a non-redundant test, we need to exercise at least one new receiver-object state. In other words, we need to explore (new) receiver-object states in order to generate non-redundant tests. In this chapter, we first present a test-generation approach that explores concrete states with method invocations (the approach was developed by us [XMN04a] and Visser et al. [VPK04] independently). Roughly this approach generates non-redundant tests only. However, this approach has two issues. First, this approach assumes that a fixed set of relevant values for method arguments are provided beforehand; supplying these relevant argument values is often a challenging task for either developers or a third-party testing tool. Second, this approach

faces a similar state exploration problem as in explicit-state model checking [CGP99].

To tackle these two issues, we have developed a test-generation approach, called Symstra, that uses symbolic execution [Kin76] of methods to explore symbolic states. Symbolic states, symbolic representations of states, describe not only single concrete states, but sets of concrete states, and when applicable, symbolic representations can yield large improvements, also previously witnessed for example by symbolic model checking [McM93]. We use symbolic execution to produce symbolic states by invoking a method with symbolic variables for primitive-type arguments, instead of requiring argument values to be provided beforehand. Each symbolic argument represents a set of all possible concrete values for the argument. We present novel techniques for comparing symbolic states of object-oriented programs. These techniques allow our Symstra approach to prune the exploration of object states and thus generate tests faster, without compromising the exhaustiveness of the exploration. In particular, the pruning preserves the intra-method path coverage of the generated test suites. We have evaluated our Symstra approach on 11 subjects, most of which are complex data structures taken from a variety of sources. The experimental results show that our Symstra approach generates tests faster than the existing concrete-state approaches [VPK04, XMN04b]. Further, given the same time for generation, our new approach can generate tests that achieve better branch coverage than the existing approaches.

The remainder of this chapter is structured as follows: Section 4.1 presents a running example. Section 4.2 describes the concrete-state approach that generates tests by exploring concrete states. Section 4.3 introduces the representation of symbolic states produced by symbolic execution. Section 4.4 presents the subsumption relationship among symbolic states and Section 4.5 introduces the Symstra approach that uses state subsumption relationship to prune symbolic-state exploration. Section 4.6 presents the experiments that we conducted to assess the approach and then Section 4.7 concludes.

4.1 Example

We use a binary search tree implementation as a running example to illustrate our Symstra approach. Figure 4.1 shows the relevant parts of the code. The binary search tree class `BST` implements a set of integers. Each tree has a pointer to the root node. Each node has an element and pointers to the

```
class BST implements Set {
    Node root;
    static class Node {
        int value;
        Node left;
        Node right;
    }
    public void add(int value) {
        if (root == null) { root = new Node(); root.value = value;  }
        else {
            Node t = root;
            while (true) {
                if (t.value < value) {        /* c1 */
                    if (t.right == null) {
                        t.right = new Node(); t.right.value = value;
                        break;
                    } else {  t = t.right;  }
                } else if (t.value > value) { /* c2 */
                    if (t.left == null) {
                        t.left = new Node(); t.left.value = value;
                        break;
                    } else {   t = t.left;  }
                } else { /* no duplicates*/   return;  }   /* c3 */
            }
        }
    }
    public void remove(int value) { ... }
    public boolean contains(int value) { ... }
}
```

Figure 4.1: A set implemented as a binary search tree

---

left and right children. The class also implements the standard set operations: `add` adds an element, if not already in the tree, to a leaf; `remove` deletes an element, if in the tree, replacing it with the smallest larger child if necessary; and `contains` checks if an element is in the tree. The class also has a default constructor that creates an empty tree.

Some tools such as Jtest [Par03] or JCrasher [CS04] test a class by generating random sequences of methods; for `BST`, they could for example generate the following tests (written in the JUnit

framework [GB03]):

```
public class BSTTest extends TestCase {
   public void test1() {
       BST t1 = new BST();
       t1.add(0);
       t1.add(-1);
       t1.remove(0);
   }

   public void test2() {
       BST t2 = new BST();
       t2.add(2147483647);
       t2.remove(2147483647);
       t2.add(-2147483648);
   }
}
```

Each test has a method sequence on the objects of the class, e.g., `test1` creates a tree `t1`, invokes two `add` methods on it, and then one `remove`. One strategy adopted by existing tools is to exhaustively explore all method sequences or randomly explore some method sequences up to a given length. These tools consider that two tests are both generated if they have different method sequences. As we have shown in Chapter 3, the conservative strategy produces a high percentage of redundant testes. The remainder of the chapter shows how to effectively generate non-redundant tests that exercise the same program behavior as exercised by those tests generated by exhaustively exploring all method sequences up to a given length.

4.2 Concrete-State Exploration

Unit-test generation for object-oriented programs consists of two parts: setting up receiver-object states and generating method arguments. The first part puts an object of the class under test into a particular state before invoking methods on it. The second part produces particular arguments for a method to be invoked on the receiver-object state. The concrete-state approach presented in this section assumes a fixed set of method arguments have been provided beforehand and invoke these method arguments to explore and set up object states.

A *method-argument state* is characterized by a method and the values for the method arguments, where a method is represented uniquely by its defining class, name, and the entire signature. Two method-argument states are equivalent iff their methods are the same and the heaps rooted from their method arguments are equivalent (isomorphic).

Each test execution produces several method executions.

Definition 9. *A* method execution $\langle s_a, s_r \rangle$ *is a pair of a method-argument state* s_a *and a receiver-object state* s_r.[1]

Then we define equivalent method executions based on equivalent states.

Definition 10. *Two method executions* $\langle s_{a1}, s_{r1} \rangle$ *and* $\langle s_{a2}, s_{r2} \rangle$ *are* equivalent *iff* s_{a1} *and* s_{a2} *are equivalent, and* s_{r1} *and* s_{r2} *are equivalent.*[2]

Our test generation approach is a type of combinatorial testing. We generate tests to exercise each possible combination of nonequivalent receiver-object states and nonequivalent method-argument states. In order to generate method-argument states, our implementation monitors and collects method arguments from the executions of existing tests. This mechanism complements existing method argument generation based on a dedicated test data pool, which contains default data values [Par03, CS04] or user-defined data values [Par03]. In practice, programmers often write unit tests [Bec00, Bec03], and these tests often contain some representative argument values. Our approach takes advantage of these tests, rather than requiring programmers to explicitly define representative argument values. When there are no manually written tests for a class, we collect method arguments exercised by tests generated by existing test-generation tools, such as Jtest [Par03] and JCrasher [CS04].

In order to prepare nonequivalent receiver-object states, initially we generate a set of tests each of which consist of only one constructor invocation. These initial tests set up "empty" receiver-object states. Then we generate new tests to exercise each nonequivalent "empty" object state with

[1]The definition of a method execution is different from the one presented in Section 3.4 of Chapter 3. This chapter represents the states of argument states and receiver states separately for the convenience of test generation, whereas Chapter 3 represents the states of argument states and receiver states in a single representation for the safety of redundant-test detection, because there may be some aliasing relationships between an argument and the receiver object, and representing them in a single representation is needed to capture these relationships conveniently.

[2]We can show that if two method executions are nonequivalent based on the preceding definition, then these two method executions are nonequivalent based on the previous definition in Section 3.4 of Chapter 3.

all nonequivalent method-argument states. After we execute the new generated tests, from the execution, we collect new object states that are not equivalent to any of those object states that have been exercised by all all nonequivalent method-argument states. Then we generate new tests to exercise each new object state with all nonequivalent method-argument states. The same iteration continues until we run of memory or time, encounter no new object state, or reach a user-specified iteration number. The iterations of generating tests are basically a process of exploring object states with method invocations in a breadth-first manner. The pseudo-code of the test-generation algorithm is presented in Figure 4.2.

The inputs to our test-generation algorithm include a set of existing tests and a user-defined maximum iteration number, which is the maximum length of method sequences in the generated tests. Our algorithm first runs the existing tests and collects runtime information, including nonequivalent constructor-argument states and nonequivalent method-argument states. We also collect the method sequence that leads to a nonequivalent object state or an argument in a method-argument state. We use these method sequences to reproduce object states or arguments.

Then for each collected nonequivalent constructor-argument state, we create a new test that invokes the constructor with the arguments. We run the new test that produces an "empty" receiver-object state. From the runtime information collected from running the new test, we determine whether the receiver-object state produced by the constructor execution is a new one (not being equivalent to any previously collected one); if so, we put it into a frontier set.

Then we iterate each object state in the frontier set and invoke each nonequivalent method-argument state on the object state. Each combination of an object state and a method argument list forms a new test. We run the new test and collect runtime information. If the receiver-object state produced by the last method execution in the new test is a new one, we put the new receiver-object state into the new frontier set for the next iteration. In the end of the current iteration, we replace the content of the current frontier set with the content of the new frontier set. We next start the subsequent iteration until we have reached the maximum iteration number or the frontier set has no object state. In the end of the algorithm, we return the generated tests collected over all iterations. These tests are exported to a test class written in the JUnit framework [GB03].

Since invoking a state-preserving method on an object state does not change the state, we can still invoke other methods on the object state in the same test. We merge generated tests as much

```
Set testGenConcreteExp(Set existingTests, int maxIterNum) {
  Set newTests = new Set();
  RuntimeInfo runtimeInfo = execAndCollect(existingTests);
  Set nonEqConstructorArgStates = runtimeInfo.getNonEqConstructorArgStates();
  Set nonEqMethodArgStates = runtimeInfo.getNonEqMethodArgStates();
  //create empty symbolic states
  Set frontiers = new Set();
  foreach (constructorArgState in nonEqConstructorArgStates) {
      Test newTest = makeTest(constructorArgState);
      newTests.add(newTest);
      runtimeInfo = execAndCollect(newTest);
      frontiers.add(runtimeInfo.getNonEqObjState());
  }
  //exercise new states from each iteration with each method-argument state
  for(int i=1;i<=maxIterNum && frontiers.size()>0;i++) {
      Set frontiersForNextIter = new Set();
      foreach (objState in frontiers) {
        foreach (argState in nonEqMethodArgStates) {
          Test newTest = makeTest(objState, argState);
          newTests.add(newTest);
          runtimeInfo = execAndCollect(newTest);
          frontiersForNextIter.add(runtimeInfo.getNonEqObjState());
        }
      }
      frontiers.clear();
      frontiers.addAll(frontiersForNextIter);
  }
  return newTests;
}
```

Figure 4.2: Pseudo-code implementation of the test-generation algorithm based on exploring concrete states.

as possible by reusing and sharing the same object states among multiple method-argument state. This reduces the number of the generated tests and the execution cost of the generated test suite. The generated test suite contains no redundant tests, since our combinatorial generation mechanism guarantees that the last method execution produced by each test is not equivalent to any method execution produced by earlier executed tests.

Our implementation uses Java reflection mechanisms [AGH00] to generate and execute new

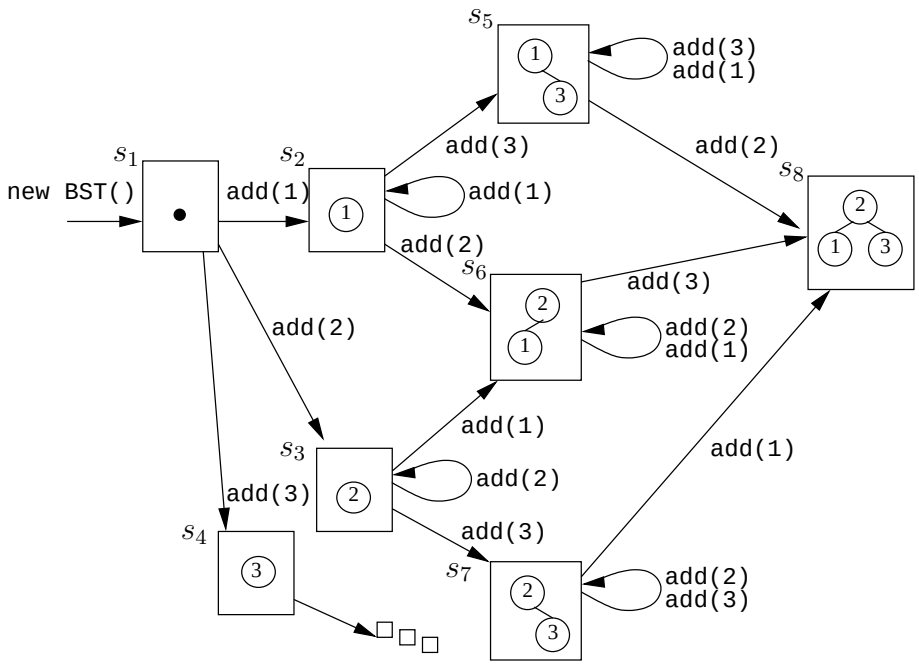

Figure 4.3: A part of the explored concrete states

tests online. In the end of test generation, we export the tests generated after each iteration to a JUnit test class code [GB03], based on JCrasher's test code generation functionality [CS04].

When we test `BST` by using the test generation algorithm in Figure 4.2, we can provide three values for `add`'s argument: `add(1)`, `add(2)`, and `add(3)`, and set the maximum iteration number as three. Figure 4.3 shows a part of the explored concrete states for the `BST` class. Each explored state has a heap, which is shown graphically in the figure. The constructor first creates an empty tree. In the first iteration, invoking `add` on the empty tree with three arguments (`1`, `2`, and `3`) produces three new states (S_2, S_3, and S_4), respectively. In the second iteration, invoking `add(1)` on S_2 does not modify the receiver-object state, still being S_2. Invoking `add(2)` and `add(3)` on S_2 produces two new states (S_5 and S_6), respectively. Similar cases occur on S_3 and S_4.

After exploring an edge (state transition), we generate a specific test to exercise this edge. We generate the test by traversing the shortest path starting from the edge of constructor invocation (`new BST()`) to the current edge, and outputting the method invocations along the path. For example, the test that we generate to exercise the edge from S_5 to S_8 is:

```
public void testEdgeFromS5ToS8() {
    BST t = new BST();
    t.add(1);
    t.add(3);
    t.add(2);
}
```

We can see that there are two major issues when we use the test generation algorithm in Figure 4.2 to test `BST`. First, the algorithm assumes that developers or third-party tools provide a set of relevant values for the method arguments. For example, if we want to generate tests to reach a `BST` object with eight elements, we need to provide at least eight different values for `add`'s argument. For complex classes, it is often a challenging task for developers or third-party tools to produce relevant values for their method arguments. Second, the algorithm faces the state explosion problem when exploring concrete states with a even relatively small number of provided method-argument values. For example, the algorithm runs out of memory when it is used to test `BST` with seven different values for the arguments of `add` and `remove` and with the maximum iteration number as seven.

In fact, invoking three `add` method invocations on the empty tree to reach S_1, S_2, and S_3 exercise the same program behavior: basically these method invocations put an integer into an empty binary search tree. Invoking `add(3)` on S_2 exercises the same program behavior as invoking `add(3)` on S_3: basically each method invocation inserts an integer into a binary search tree containing a smaller integer. To tackle the state exploration problem, we can construct an abstraction function that maps similar concrete states into a single abstract state. One challenge here is to construct this abstraction function automatically. The next section presents our new approach, called Symstra, that uses symbolic execution to automatically group several concrete states into a single symbolic state, if these concrete states are isomorphic in an abstract level and they are reached by executing the same path of the program.

4.3 Symbolic-State Representation

The symbolic execution [Kin76] of a method accepts method inputs in the form of *symbolic variables*, instead of actual arguments values. In the symbolic execution of an object-oriented program, the receiver object of a method invocation can be in *symbolic states*. Symbolic states differ from

concrete states, on which the usual program executions operate, in that symbolic states contain a symbolic heap that includes symbolic expressions with symbolic variables (such as symbolic variables connected with their associated types' operators), and contain also constraints on these variables.

We view a symbolic heap as a graph: nodes represent objects (as well as primitive values and symbolic expressions) and edges represent object fields. Let O be some set of objects whose fields form a set F. Each object has a field that represents its class. We consider arrays as objects whose fields are labelled with (integer) array indexes and point to the array elements.

Definition 11. *A symbolic heap is an edge-labelled graph $\langle O, E\rangle$, where $E \subseteq O \times F \times (O \cup \{\texttt{null}\} \cup U)$ such that for each field f of each $o \in O$ exactly one $\langle o, f, o'\rangle \in E$. A concrete heap has only concrete values: $o' \in O \cup \{\texttt{null}\} \cup P$.*

Given the definition of a symbolic heap, we can then define a symbolic state formally:

Definition 12. *A* symbolic state $\langle C, H\rangle$ *is a pair of a constraint and a symbolic heap.*

The usual execution of a method starts with a concrete state of the receiver object and method-argument values, and then produces one return value and one concrete state of the receiver object. In contrast, the symbolic execution of a method starts with a symbolic state of the receiver object and symbolic variables of method arguments, and then produces several return values and several symbolic states of the receiver object. A *symbolic execution tree* characterizes the execution paths followed during the symbolic execution of a program. An edge represents a method invocation whose symbolic execution follows a specific path. A node in the tree represents a symbolic state produced by symbolically executing a specific path of a method. Figure 4.4 shows a part of the symbolic execution tree for `BST` when we invoke a method sequence consisting of only the `add` method.

The constructor of `BST` first creates an empty tree S_1, whose constraint is `true`. Then we invoke `add` on S_1 with symbolic variable x_1 as the argument. The symbolic execution of `add` on S_1 can explore one path, producing a symbolic state S_2 whose heap contains the element x_1 and constraint is still `true`. In general, while an execution of a method with concrete arguments produces one state, the symbolic execution of a method with symbolic arguments can produce several states, thus

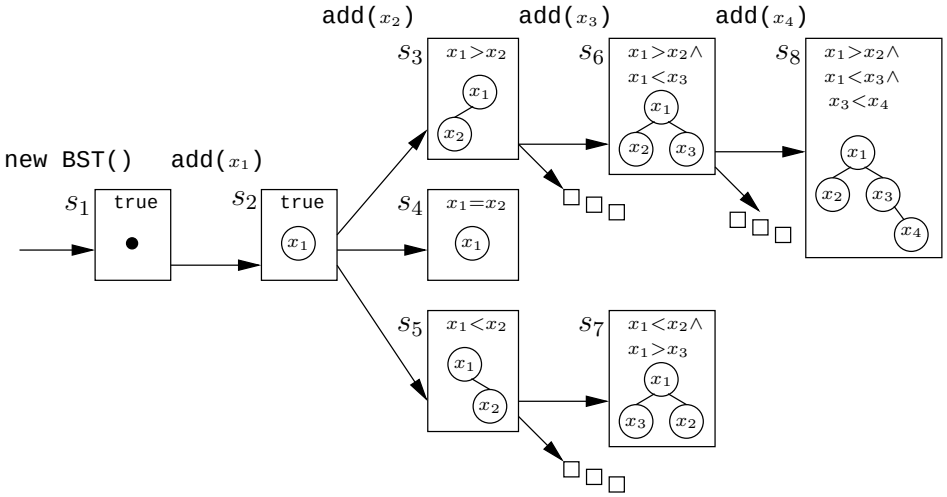

Figure 4.4: A part of the symbolic execution tree

resulting in an execution tree. For example, the symbolic execution of the `add` on S_2 with symbolic variable x_2 as the argument produces three symbolic states (S_3, S_4, and S_5), which are produced by following three different paths within `add`, in particular, taking three different branches ($c1$, $c2$, and $c3$) labeled in the method body of `add` (Figure 4.1): if $x_1 = x_2$, the tree does not change, and if $x_2 > x_1$ (or $x_2 < x_1$), x_2 is added in the right (or left) subtree.

Following the typical symbolic executions [Kin76, KPV03, VPK04], our implementation symbolically explores both branches of `if` statements, modifying the constraint with a conjunct that needs to hold for the execution to take a certain branch. In this context, the constraint is called *path condition*, because it is a conjunction of conditions that need to hold for the execution to take a certain path and reach the current address. This symbolic execution directly explores every path of the method under consideration. The common issue in the symbolic execution is that the number of paths may be infinite (or too large as it grows exponentially with the number of branches). In such cases, we can use the standard set of heuristics to explore only some of the paths [VPK04, BPS00].

Our implementation executes code on symbolic states by rewriting the code to operate on symbolic expressions [KPV03, VPK04]. Furthermore, Symstra implements the exploration of different branches by re-executing the method from the beginning for each path, without storing any intermediate states. Note that Symstra re-executes only one method (for different paths), not the whole

method sequence. This effectively produces a depth-first exploration of paths within one method, while the exploration of states between methods is breadth-first as explained in the next section. Our Symstra prototype also implements the standard optimizations for symbolic execution. First, Symstra simplifies the constraints that it builds at branches; specifically, before conjoining the path condition so far C and the current branch condition C' (where C' is a condition from an `if` or its negation), Symstra checks if some of the conjuncts in C implies C'; if so, Symstra does not conjoin C'. Second, Symstra checks if the constraint $C \&\& C'$ is unsatisfiable; if so, Symstra stops the current path of symbolic execution, because it is an infeasible path. The current Symstra prototype can use the Simplify [DNS03] theorem prover or the Omega library [Pug92] to check unsatisfiability.

4.4 Symbolic-State Subsumption

This section presents techniques that compare two symbolic states: checking isomorphism of their symbolic heaps and checking implication relationships between their constraints. These techniques help determine symbolic-state subsumption: whether one symbolic state subsumes the other. We use symbolic-state subsumption to effectively prune the exploration of symbolic states (Section 4.5).

4.4.1 Heap-Isomorphism Checking

We define heap isomorphism as graph isomorphism based on node bijection [BKM02]. We want to detect isomorphic heaps because invoking the same methods on them leads to equivalent method behaviors and redundant tests; therefore, it suffices to explore only one representative from each isomorphism partition. Nodes in symbolic heaps contain symbolic variables, so we first define a *renaming* of symbolic variables. Given a bijection $\tau : V \to V$, we extend it to the whole $\tau : U \to U$ as follows: $\tau(p) = p$ for all $p \in P$, and $\tau(\odot u_1, \ldots, u_n) = \odot \tau(u_1), \ldots, \tau(u_n)$ for all $u_1, \ldots, u_n \in U$ and operations $\odot$. We further extend τ to substitute free variables in formulas with bound variables, avoiding capture as usual.

Definition 13. *Two symbolic heaps $\langle O_1, E_1 \rangle$ and $\langle O_2, E_2 \rangle$ are* isomorphic *iff there are bijections*

$\rho : O_1 \to O_2$ and $\tau : V \to V$ such that:

$$\begin{aligned} E_2 \;=\; & \{\langle \rho(o), f, \rho(o')\rangle | \langle o, f, o'\rangle \in E_1, o' \in O_1\} \cup \{\langle \rho(o), f, \texttt{null}\rangle | \langle o, f, \texttt{null}\rangle \in E_1\} \cup \\ & \{\langle \rho(o), f, \tau(o')\rangle | \langle o, f, o'\rangle \in E_1, o' \in U\}. \end{aligned}$$

The definition allows only object identities and symbolic variables to vary: two isomorphic heaps have the same fields for all objects and equal (up to renaming) symbolic expressions for all primitive fields.

Our test generation based on state exploration does not consider the entire program heap but focuses on the state of several objects (including the receiver object and arguments of a method invocation); in this context, the state of an object o is a rooted heap, which is characterized by the values of the fields of o and fields of all objects *reachable* from o.

We *linearize* rooted symbolic heaps into integer sequences such that checking symbolic-heap isomorphism corresponds to checking sequence equality. Figure 4.5 shows the linearization algorithm for a symbolic rooted heap. It starts from the root and traverses the heap in a depth-first manner. It assigns a unique identifier to each object that is visited for the first time, keeps this mapping in `objs`, and reuses it for objects that appear in cycles. It also assigns a unique identifier to each symbolic variable, keeps this mapping in `vars`, and reuses it for variables that appear several times in the heap.

This algorithm extends the linearization algorithm shown in Figure 3.2 of Chapter 3 with `linSymExp` that handles symbolic expressions; this improves on the approach of Khurshid et al. [KPV03, VPK04] that does not use any comparison for symbolic expressions. We can show that our linearization normalizes rooted heaps.

Theorem 2. *Two rooted heaps $\langle O_1, E_1\rangle$ (with root r_1) and $\langle O_2, E_2\rangle$ (with root r_2) are isomorphic iff* `linearize`$(r_1, \langle O_1, E_1\rangle)$=`linearize`$(r_2, \langle O_2, E_2\rangle)$.

4.4.2 State-Subsumption Checking

When the rooted heaps in two symbolic states are isomorphic, these two symbolic states are not necessarily equivalent (based on the observational equivalence [DF94, LG00]), because the constraints in these two symbolic states may not be equivalent (two constraints are equivalent if they

```
Map<Object,int> objs; // maps objects to unique ids
Map<SymVar,int> vars; // maps symbolic variables to unique ids

int[] linearize(Object root, Heap <O,E>) {
  objs = new Map();  vars = new Map();
  return lin(root, <O,E>>;
}

int[] lin(Object root, Heap <O,E>) {
  if (objs.containsKey(root))
    return singletonSequence(objs.get(root));
  int id = objs.size() + 1;  objs.put(root, id);
  int[] seq = singletonSequence(id);
  Edge[] fields = sortByField({ <root, f, o> in E });
  foreach (<root, f, o> in fields) {
    if (isSymbolicExpression(o)) seq.append(linSymExp(o));
    elseif (o == null) seq.append(0);
    else seq.append(lin(o, <O,E>)); // pointer to an object
  }
  return seq;
}

int[] linSymExp(SymExp e) {
  if (isSymVar(e)) {
    if (!vars.containsKey(e))
      vars.put(e, vars.size() + 1);
    return singletonSequence(vars.get(e));
  } elseif (isPrimitive(e)) return uniqueRepresentation(e);
  else { // operation with operands
    int[] seq = singletonSequence(uniqueRepresentation(e.getOperation()));
    foreach (SymExp e' in e.getOperands())
      seq.append(linSymExp(e'));
    return seq;
  }
}
```

Figure 4.5: Pseudo-code of linearization for a symbolic rooted heap

have the same set of solutions). Two symbolic states are equivalent if they represent the same set of concrete states. To effectively prune the exploration of symbolic states, we define the subsumption

```
boolean checkSubsumes(Constraint C1, Heap H1,
                      Constraint C2, Heap H2) {
    int[] i1 = linearize(root(H1), H1);
    Map<SymVar,int> v1 = vars; // at the end of previous linearization
    Set<SymVar> n1 = variables(C1) - v1.keys(); // variables not in the heap
    int[] i2 = linearize(root(H2), H2);
    Map<SymVar,int> v2 = vars; // at the end of previous linearization
    Set<SymVar> n2 = variables(C2) - v2.keys(); // variables not in the heap
    if (i1 <> i2) return false;
    Renaming τ = v2 ∘ v1^-1 // compose v2 and the inverse of v1
    return checkValidity(τ(∃n2. C2) ⇒ ∃n1. C1);
}
```

Figure 4.6: Pseudo-code of subsumption checking for symbolic states

relationships among symbolic states. Intuitively a symbolic state S subsumes another one S' if the concrete states represented by S are a superset of the concrete states represented by S'; then if we have explored S, we do not need to explore S', because the behaviors exercised by invoking methods on S' would have been exercised by invoking methods on S. We can more effectively prune the exploration of symbolic states based on symbolic-state subsumption than based on symbolic-state equivalence.

We next formally define symbolic state subsumption based on the concrete heaps that each symbolic state represents. To instantiate a symbolic heap into a concrete heap, we replace the symbolic variables in the heap with primitive values that satisfy the constraint in the symbolic state.

Definition 14. *An* instantiation $\mathcal{I}(\langle C, H\rangle)$ *of a symbolic state* $\langle C, H\rangle$ *is a set of concrete heaps* H' *such that there exists a valuation* $\eta : V \to P$ *for which* $\eta(C)$ *is true and* H' *is the evaluation* $\eta(H)$ *of all expressions in* H *according to* η*.*

Definition 15. *A symbolic state* $\langle C_1, H_1\rangle$ subsumes *another symbolic state* $\langle C_2, H_2\rangle$*, in notation* $\langle C_1, H_1\rangle \supseteq \langle C_2, H_2\rangle$*, iff for each concrete heap* $H'_2 \in \mathcal{I}(\langle C_2, H_2\rangle)$*, there exists a concrete heap* $H'_1 \in \mathcal{I}(\langle C_1, H_1\rangle)$ *such that* H'_1 *and* H'_2 *are isomorphic.*

We use the algorithm in Figure 4.6 to check if the constraint of $\langle C_2, H_2\rangle$, after suitable renaming, implies the constraint of $\langle C_1, H_1\rangle$. When some symbolic variables are removed from the heaps, for

example, by a `remove` method, these symbolic variables do not appear in the heaps but may appear in a constraint. Therefore, the implication is universally quantified over only the (renamed) symbolic variables that appear in the heaps and existentially quantified over the symbolic variables that do not appear in the heaps (more precisely only in H_1, because the existential quantifier for n_2 in the premise of the implication becomes a universal quantifier for the whole implication).

We can show that this algorithm is a conservative approximation of subsumption.

Theorem 3. *If* `checkSubsumes`*(*$\langle C_1, H_1\rangle, \langle C_2, H_2\rangle$*) then* $\langle C_1, H_1\rangle$ *subsumes* $\langle C_2, H_2\rangle$.

For example, we can show that the heaps in S_2 and S_4 (Figure 4.4) are isomorphic and the implication $(\forall x_1 \exists x_2 (x_1 = x_2) \Rightarrow true)$ holds. Then we can determine S_2 subsumes S_4. Similarly we can determine S_6 subsumes S_7. Note that the renaming operation on constraints (shown in Figure 4.6) is necessary for us to show that the constraint of S_7 implies the constraint of S_6.

Our Symstra approach gains the power and inherits the limitations from the technique used to check the implication on the (renamed) constraints. Our implementation uses the Omega library [Pug92], which provides a complete decision procedure for Presburger arithmetic, and CVC Lite [BB04], an automatic theorem prover, which has decision procedures for several types of constraints, including real linear arithmetic, uninterpreted functions, arrays, etc. Because these checks can consume a lot of time, our implementation further uses the following conservative approximation: if free-variables($\exists n_1.\ C_1$) are not a subset of free-variables($\tau(\exists n_2.\ C_2)$), return `false` without checking the implication.

4.5 Symbolic-State Exploration

We next present how our Symstra approach systematically explores the symbolic-state space. The state space consists of all symbolic states that are reachable with the symbolic execution of a method for the class under test. Our Symstra approach exhaustively explores a bounded part of the symbolic state space using a breadth-first search. The pseudo-code of the test-generation algorithm is presented in Figure 4.7.

The inputs to our test-generation algorithm include a set of constructor $\mathcal{C}$ and non-constructor methods $\mathcal{M}$ of the class under test, and a user-defined maximum iteration number, which is the maximum length of method sequences in the generated tests. We first invoke each constructor on

```
Set testGenSymExp(Set C, Set M, int maxIterNum) {
  Set newTests = new Set();
  //create empty symbolic states
  Set frontiers = new Set();
  foreach (constructor in C) {
      RuntimeInfo runtimeInfo = symExecAndCollect(constructor);
      newTests.addAll(runtimeInfo.solveAndGenTests());
      frontiers.addAll(runtimeInfo.getNonSubsumedObjStates());
  }
  //exercise non-subsumed symbolic states with symbolic execution of methods
  for(int i=1;i<=maxIterNum && frontiers.size()>0;i++) {
      Set frontiersForNextIter = new Set();
      foreach (objState in frontiers) {
        foreach (method in M) {
          RuntimeInfo runtimeInfo = symExecAndCollect(objState, method);
          newTests.addAll(runtimeInfo.solveAndGenTests());
          frontiersForNextIter.addAll(runtimeInfo.getNonSubsumedObjStates());
        }
      }
      frontiers.clear();
      frontiers.addAll(frontiersForNextIter);
  }
  return newTests;
}
```

Figure 4.7: Pseudo-code implementation of the test-generation algorithm based on exploring symbolic states.

---

the initial symbolic state, which is $s_0 = \langle \texttt{true}, \{\} \rangle$: the constraint is true, and the heap is empty. The symbolic execution of the constructor produces some "empty" receiver-object states. Then for each symbolic state produced by the symbolic execution, we generate a test. We also determine whether the symbolic state is subsumed by any previously collected symbolic state; if not, we collect it into a frontier set.

Then we iterate each symbolic-object state collected in the frontier set and invoke each method in $\mathcal{M}$ on the object state. We create a new test for each symbolic state S produced by the symbolic execution of the method. If S is not subsumed by any previously collected symbolic state, we collect S into the new frontier set for the next iteration. Otherwise, we prune the further exploration of S: S

represents only a subset of the concrete heaps that are represented by some symbolic state previously collected for exploration; it is thus unnecessary to explore S further. Pruning based on subsumption plays the key role in enabling our algorithm to explore large state spaces. For example, S_4 and S_7 in Figure 4.4 are pruned because we have collected and explored S_2 and S_6, which subsume S_4 and S_7, respectively.

In the end of the current iteration, we replace the content of the current frontier set with the content of the new frontier set. We next start the subsequent iteration until we have reached the maximum iteration number or the frontier set has no symbolic state. In the end of the algorithm, we return the generated tests collected over all iterations. These tests are exported to a test class written in the JUnit framework [GB03].

During the symbolic-state exploration, we build specific concrete tests that lead to the states explored through the symbolic execution of a method. Whenever we finish a method m's symbolic execution that generates a symbolic state $\langle C, H \rangle$, we first generate a *symbolic test*, which consists of the constraint C and the sequence of method invocations along the shortest path starting from the edge of constructor invocation to the edge for m's symbolic execution. We then instantiate the symbolic test using the POOC constraint solver [SR02] to solve the constraint C over the symbolic arguments for methods in the sequence. Based on the produced solution, we obtain concrete arguments for the sequence leading to $\langle C, H \rangle$. We export such concrete test sequences into a JUnit test class [GB03]. We also export the constraint C associated with the test as a comment for the test in the JUnit test class.

For example, the tests that we generate to exercise the edge from S_2 to S_3 and the edge from S_2 to S_5 in Figure 4.4 are:

```
public void testEdgeFromS2ToS3() {
    /* x1 > x2 */
    int x1 = -999999;
    int x2 = -1000000;
    BST t = new BST();
    t.add(x1);
    t.add(x2);
}

public void testEdgeFromS2ToS5() {
    /* x1 < x2 */
```

```
        int x1 = -1000000;
        int x2 = -999999;
        BST t = new BST();
        t.add(x1);
        t.add(x2);
    }
```

A realistic suite of unit tests contains more sequences that test the interplay between `add`, `remove`, and `contains` methods. Section 4.6 summarizes such suites.

At the class-loading time, our implementation instruments each branching point of the class under test for measuring branch coverage at the bytecode level. It also instruments each method of the class to capture uncaught exceptions at runtime. Given a symbolic state at the entry of a method execution, our implementation uses symbolic execution to achieve structural coverage within the method, because symbolic execution systematically explores all feasible paths within the method. If the user of Symstra is interested in only the tests that achieve new branch coverage, our implementation selects only the generated tests that increase branch coverage or throw new uncaught exceptions. Our implementation can also be extended for selecting tests that achieve new bounded intra-method path coverage [BL00].

4.6 Evaluation

This section presents our evaluation of Symstra for exploring states and generating tests. We compare Symstra with the concrete-state approach shown in Section 4.2. We have developed both approaches within the same infrastructure, so that the comparison does not give an unfair advantage to either approach because of unrelated improvements. In these experiments, we have used the Simplify [DNS03] theorem prover to check unsatisfiability of path conditions, the Omega library [Pug92] to check implications, and the POOC constraint solver [SR02] to solve constraints. We have performed the experiments on a Linux machine with a Pentium IV 2.8 GHz processor using Sun's Java 2 SDK 1.4.2 JVM with 512 MB allocated memory.

Table 4.1 lists the 11 Java classes that we use in the experiments. The first six classes were previously used in evaluating our redundant-test detection approach presented in Chapter 3, and the last five classes were used in evaluating Korat [BKM02]. The columns of the table show the class

Table 4.1: Experimental subjects

class	methods under test	some private methods	#ncnb lines	# branches
IntStack	push,pop	–	30	9
UBStack	push,pop	–	59	13
BinSearchTree	add,remove	removeNode	91	34
BinomialHeap	insert,extractMin delete	findMin,merge unionNodes,decrease	309	70
LinkedList	add,remove,removeLast	addBefore	253	12
TreeMap	put,remove	fixAfterIns fixAfterDel,delEntry	370	170
HeapArray	insert,extractMax	heapifyUp,heapifyDown	71	29

name, the public methods under test (that the generated sequences consist of), some private methods invoked by the public methods, the number of non-comment, non-blank lines of code in all those methods, and the number of branches for each subject.

We use both approaches to explore states up to N iterations; in other words, we generate tests that consist of sequences with up to N methods. The concrete-state approach also requires concrete values for arguments, so we set it to use N different arguments (the integers from 0 to $N - 1$) for methods under test. Table 4.2 shows the comparison between Symstra and the concrete-state approach. We consider N in the range from five to eight. (For $N < 5$, both approaches generate tests really fast, usually within a couple of seconds, but those tests do not have good quality.) We tabulate the time to generate the tests (measured in seconds, Columns 3 and 7), the number of explored symbolic and concrete object states (Columns 4 and 8), the number of generated tests (Columns 5 and 9), and the branch coverage[3] achieved by the generated tests (Columns 6 and 10). In Columns 5 and 9, we report the total number of generated tests and, in the parentheses, the cumulative number of tests that increase the branch coverage.

During test generation, we set a three-minute timeout for each iteration of the breadth-first ex-

[3]We measure the branch coverage at the bytecode level during the state exploration of both approaches, and calculate the total number of branches also at the bytecode level.

ploration: when an iteration exceeds three minutes, the exhaustive exploration of each approach is stopped and the system proceeds with the next iteration. We use a "*" mark for each entry where the test-generation process timed out; the state exploration of these entries is no longer exhaustive. We use a "–" mark for each entry where its corresponding approach exceeded the memory limit.

The results indicate that Symstra generates method sequences of the same length N often much faster than the concrete-state approach, thus enabling Symstra to generate longer method sequences within a given time limit. Both approaches achieve the same branch coverage for method sequences of the same length N. However, Symstra achieves higher coverage faster. It also takes less memory and can finish generation in more cases. These results are due to the fact that each symbolic state, which Symstra explores at once, actually describes a set of concrete states, which the concrete-state approach must explore one by one. The concrete-state approach often exceeds the memory limit when $N = 7$ or $N = 8$, which is often not enough to guarantee full branch coverage.

4.7 Conclusion

We have proposed Symstra, an approach that uses symbolic execution to generate a small number of non-redundant tests that achieve high branch and intra-method path coverage for complex data structures. Symstra exhaustively explores symbolic states with symbolic arguments up to a given length. It prunes the exploration based on state subsumption; this pruning speeds up the exploration, without compromising its exhaustiveness. We have implemented the approach and evaluated it on 11 subjects, most of which are complex data structures. The results show that Symstra generates tests faster than the existing concrete-state approaches, and given the same time limit, Symstra can generate tests that achieve better branch coverage than these existing approaches.

We finally discuss how Symstra can be leveraged in specification-based testing, and extended to improve performance and address some inherent limitations of symbolic execution.

Specifications. Although the work in this dissertation including the Symstra approach has been developed to be used in the absence of specifications, Symstra's test generation can be guided by specifications if they are provided. These specifications can include method pre- and post-conditions and class invariants, written in the Java Modelling Language (JML) [LBR98]. The JML tool-set

transforms these constructs into run-time assertions that throw JML-specific exceptions when violated. Specification-based testing normally needs to generate *legal* method invocations whose method-entry states satisfy pre-conditions and class invariants, i.e., no exceptions for these constructs are thrown at method entries. By default, Symstra does not explore further a state resulting from an exception-throwing method execution; therefore, Symstra explores legal method sequences. If during the exploration Symstra finds a method invocation that violates a post-condition or invariant, Symstra has discovered a bug; Symstra can be configured to generate such tests and continue or stop test generation. If a class implementation is correct with respect to its specification, paths that throw post-condition or invariant exceptions should be infeasible.

Our implementation for Symstra operates on the bytecode level. It can perform testing of the specifications woven into method bytecode by the JML tool-set or by similar tools. Note that in this setting Symstra essentially uses black-box testing [VPK04] to explore only those symbolic states that are produced by method executions that satisfy pre-conditions and class invariants; conditions that appear in specifications simply propagate into the constraints associated with a symbolic state explored by Symstra. Using symbolic execution, Symstra thus obtains the generation of legal test sequences "for free".

Performance. Based on state subsumption, our current implementation for Symstra explores one or more symbolic states that have the isomorphic heap. We can extend our implementation to explore exactly one *union* symbolic state for each isomorphic heap. We can create a union state using a disjunction of the constraints for all symbolic states with the isomorphic heap. Each union state subsumes all the symbolic states with the isomorphic heap, and thus exploring only union states can further reduce the number of explored states without compromising the exhaustiveness of the exploration. (Subsumption is a special case of union; if $C_2 \Rightarrow C_1$, then $C_1 \vee C_2$ simplifies to C_1.)

Symstra enables exploring longer method sequences than the concrete-state approaches. However, users may want to have an exploration of even longer sequences to achieve some test purpose. In such cases, the users can apply several techniques that trade the guarantee of the intra-method path coverage for longer sequences. For example, the users may provide abstraction functions for states [LG00], as used for instance in the AsmLT generation tool [Fou], or binary methods for com-

paring states (e.g. `equals`), as used for instance in our Rostra approach (Chapter 3). Symstra can then generate tests that instead of subsumption use these user-provided functions for comparing state. This leads to a potential loss of intra-method path coverage but enables faster, user-controlled exploration. To explore longer sequences, Symstra can also use standard heuristics [VPK04, BPS00] for selecting only a set of paths instead of exploring all paths.

Limitations. The use of symbolic execution has inherent limitations. For example, it cannot precisely handle array indexes that are symbolic variables. This situation occurs in some classes, such as `DisjSet` and `HashMap` used previously in evaluating Rostra (Chapter 3). One solution is to combine symbolic execution with (exhaustive or random) exploration based on concrete arguments: a static analysis would determine which arguments can be symbolically executed, and for the rest, the user would provide a set of concrete values [Fou].

So far we have discussed only methods that take primitive arguments. We cannot directly transform non-primitive arguments into symbolic variables of primitive type. However, we can use the standard approach for generating non-primitive arguments: generate them also as sequences of method calls that may recursively require more sequences of method calls, but eventually boil down to methods that have only primitive values (or `null`). (Note that this also handles mutually recursive classes.) JCrasher [CS04] and Eclat [PE05] take a similar approach. Another solution is to transform these arguments into reference-type symbolic variables and enhance the symbolic execution to support heap operations on symbolic references. Concrete objects representing these variables can be generated by solving the constraints and setting the instance fields using reflection. However, the collected constraints are often not sufficient to generate legal instances, in which case an additional object invariant is required.

Table 4.2: Experimental results of test generation using Symstra and the concrete-state approach

		Symstra				Concrete-State Approach			
class	N	time	states	tests	%cov	time	states	tests	%cov
UBStack	5	0.95	22	43(5)	92.3	4.98	656	1950(6)	92.3
	6	4.38	30	67(6)	100.0	31.83	3235	13734(7)	100.0
	7	7.20	41	91(6)	100.0	*269.68	*10735	*54176(7)	*100.0
	8	10.64	55	124(6)	100.0	-	-	-	-
IntStack	5	0.23	12	18(3)	55.6	12.76	4836	5766(4)	55.6
	6	0.42	16	24(4)	66.7	-	-	-	-
	7	0.50	20	32(5)	88.9	*689.02	*30080	*52480(5)	*66.7
	8	0.62	24	40(6)	100.0	-	-	-	-
BinSearchTree	5	7.06	65	350(15)	97.1	4.80	188	1460(16)	97.1
	6	28.53	197	1274(16)	100.0	23.05	731	7188(17)	100.0
	7	136.82	626	4706(16)	100.0	-	-	-	-
	8	*317.76	*1458	*8696(16)	*100.0	-	-	-	-
BinomialHeap	5	1.39	6	40(13)	84.3	4.97	380	1320(12)	84.3
	6	2.55	7	66(13)	84.3	50.92	3036	12168(12)	84.3
	7	3.80	8	86(15)	90.0	-	-	-	-
	8	8.85	9	157(16)	91.4	-	-	-	-
LinkedList	5	0.56	6	25(5)	100.0	32.61	3906	8591(6)	100.0
	6	0.66	7	33(5)	100.0	*412.00	*9331	*20215(6)	*100.0
	7	0.78	8	42(5)	100.0	-	-	-	-
	8	0.95	9	52(5)	100.0	-	-	-	-
TreeMap	5	3.20	16	114(29)	76.5	3.52	72	560(31)	76.5
	6	7.78	28	260(35)	82.9	12.42	185	2076(37)	82.9
	7	19.45	59	572(37)	84.1	41.89	537	6580(39)	84.1
	8	63.21	111	1486(37)	84.1	-	-	-	-
HeapArray	5	1.36	14	36(9)	75.9	3.75	664	1296(10)	75.9
	6	2.59	20	65(11)	89.7	-	-	-	-
	7	4.78	35	109(13)	100.0	-	-	-	-
	8	11.20	54	220(13)	100.0	-	-	-	-

Chapter 5

TEST SELECTION FOR INSPECTION

In practice, developers tend to write a relatively small number of unit tests, which in turn tend to be useful but insufficient for high software quality assurance. Some automatic test-generation tools, such as Parasoft Jtest [Par03], attempt to fill the gaps not covered by any manually generated unit tests; these tools can automatically generate a large number of unit test inputs to exercise the program. However, there are often no expected outputs (oracles) for these automatically generated test inputs and the tools generally only check the program's robustness: checking whether any uncaught exception is thrown during test executions [KJS98, CS04]. Manually verifying the outputs of such a large number of test inputs requires intensive labor, which is usually impractical. Unit-test selection is a means to address this problem by selecting the most valuable subset of the automatically generated test inputs. Then programmers can inspect the executions of this much smaller set of test inputs to check the correctness or robustness, and to add oracles.

If *a priori* specifications are provided with a program, the execution of automatically generated test inputs can be checked against the specifications to determine the correctness. In addition, specifications can guide test generation tools to generate test inputs. For example, the preconditions in specifications can guide test generation tools to generate only valid test inputs that satisfy the preconditions [Par03, BKM02]. The postconditions in specifications can guide test generation tools to generate test inputs to try to violate the postconditions, which are fault-exposing test inputs [Par03, KAY96, Gup03]. Although specifications can bring us many benefits in testing, specifications often do not exist in practice.

We have developed the *operational violation* approach: a black-box test generation and selection approach that does not require *a priori* specifications. An *operational abstraction* describes the actual behavior during program execution of an existing unit test suite [HME03]. We use the generated operational abstractions to guide test generation tools, so that the tools can more effectively generate test inputs that violate these operational abstractions. If the execution of an automati-

cally generated test input violates an operational abstraction, we select this test input for inspection. The key idea behind this approach is that the violating test exercises a new feature of program behavior that is not covered by the existing test suite. We have implemented this approach by integrating Daikon [ECGN01] (a dynamic invariant detection tool) and the commercial Parasoft Jtest 4.5 [Par03].

The next section describes the example that we use to illustrate our approach. Section 5.2 presents the operational violation approach. Section 5.3 describes the experiments that we conducted to assess the approach and then Section 5.4 concludes.

5.1 Example

This section presents an example to illustrate how programmers can use our approach to test their programs. The example is a Java implementation `UBStack` of a bounded stack that stores unique elements of integer type. Figure 5.1 shows the class including several method implementations that we shall refer to in the rest of the chapter. Stotts et al. coded this Java implementation to experiment with their algebraic-specification-based approach for systematically creating unit tests [SLA02]; they provided a web link to the full source code and associated test suites. Stotts et al. also specified formal algebraic specifications for the bounded stack.

In the class implementation, the array field `elems` contains the elements of the stack, and the integer field `numberOfElements` is the number of the elements and the index of the first free location in the stack. The integer field `max` is the capacity of the stack. The `pop` method simply decreases `numberOfElements`. The `top` method returns the element in the array with the index of `numberOfElements-1` if `numberOfElements >= 0`. Otherwise, the method prints an error message and returns `-1` as an error indicator. The `getSize` method returns `numberOfElements`. Given an element, the `isMember` method returns `true` if it finds the same element in the subarray of `elems` up to `numberOfElements`, and returns `false` otherwise.

Stotts et al. have created two unit test suites for this class: a basic JUnit [GB03] test suite (8 tests), in which one test method is generated for a public method in the target class; and a JAX test suite (16 tests), in which one test method is generated for an axiom in `UBStack`'s algebraic specifications. The basic JUnit test suite does not expose any fault but one of the JAX test cases exposes

```
public class UBStack {
  private int[] elems;
  private int numberOfElements;
  private int max;
  public UBStack() {
    numberOfElements = 0;
    max = 2;
    elems = new int[max];
  }
  public void push(int k) { ...  }
  public void pop() {  numberOfElements--;  }
  public int top() {
    if (numberOfElements < 1) {
      System.out.println("Empty Stack");
      return -1;
    } else {
      return elems[numberOfElements-1];
    }
  }
  public int getSize() {  return numberOfElements; }
  public boolean isMember(int k) {
    for(int index=0; index<numberOfElements; index++)
      if (k==elems[index])
        return true;
    return false;
  }
  ...
}
```

Figure 5.1: The `UBStack` program

---

one fault (handling a `pop` operation on an empty stack incorrectly). In practice, programmers usually fix the faults exposed by the existing unit tests before they augment the unit test suite. In this example and for our analysis of our approach, instead of fixing the exposed fault, we remove this fault-revealing test case from the JAX test suite to make all the existing test cases pass.

5.2 Operational Violation Approach

In this work, the objective of *unit-test selection* is to select the most valuable subset of automatically generated tests for inspection and then use these selected tests to augment the existing tests for a program unit. More precisely, we want to select generated tests to exercise a program unit's new behavior that is not exercised by the existing tests. Since manual effort is required to verify the results of selected test inputs, it is important to select a relatively small number of tests. This is different from the problems that traditional test selection techniques address [CR99, HME03]. In those problems, there are test oracles for unselected test inputs. Therefore, selecting a relatively large number of tests during selection is usually acceptable for those problems, but is not practical in this work. More formally, the objective of unit-test selection in this context is to answer the following question as inexpensively as possible:

Problem. *Given a program unit u, a set S of existing tests for u, and a test t from a set S' of generated tests for u, does the execution of t exercise at least one new feature that is not exercised by the execution of any test in S?*

If the answer is yes, `t` is removed from `S'` and put into `S`. Otherwise, `t` is removed from `S'` and discarded. In this work, the initial set `S` comprises the existing unit tests, which are usually manually written. The set `S'` of unselected tests is automatically generated tests.

The term *feature* is intentionally vague, since it can be defined in different ways. For example, a new feature could be fault-revealing behavior that does not occur during the execution of the existing tests. A new feature could be a predicate in the specifications for the unit [CR99]. A new feature could be program behavior exhibited by executing a new structural entity, such as statement, branch, or def-use pair.

Our operational violation approach uses operational abstractions to characterize program features. An *operational abstraction* is a collection of logical statements that abstract the program's runtime behavior [HME03]. It is syntactically identical to a formal specification. In contrast to a formal specification, which expresses desired behavior, an operational abstraction expresses observed behavior. Daikon [ECGN01], a dynamic invariant detection tool, can be used to infer operational abstractions (also called invariants) from program executions of test suites. These operational abstractions are in the form of DbC annotations [Mey92, LBR98, Par02]. Daikon examines variable

values computed during executions and generalizes over these values to obtain operational abstractions. Like other dynamic analysis techniques, the quality of the test suite affects the quality of the analysis. Deficient test suites or a subset of sufficient test suites may not help to infer a generalizable program property. Nonetheless, operational abstractions inferred from the executed test suites constitute a summary of the test execution history. In other words, the executions of the test suites all satisfy the properties in the generated operational abstractions.

Our approach leverages an existing specification-based test generation tool to generate new tests and selects those generated tests that violate the operational abstractions inferred from the existing tests. Our implementation uses Parasoft Jtest 4.5 [Par03]. Jtest can automatically generate unit tests for a Java class. When specifications are provided with the class, Jtest can make use of them to perform black-box testing. The provided preconditions, postconditions, or class invariants give extra guidance to Jtest in its test generation. If the code has preconditions, Jtest tries to generate test inputs that satisfy all of them. If the code has postconditions, Jtest generates test inputs that verify whether the code satisfies these conditions. If the code has class invariants, Jtest generates test inputs that try to make them fail. By default, Jtest tests each method by generating arguments for them and calling them independently. In other words, Jtest basically tries the calling sequences of length one by default. Tool users can set the length of calling sequences in the range of one to three. If a calling sequence of length three is specified, Jtest first tries all calling sequences of length one followed by all those of length two and three sequentially.

Section 5.2.1 next explains the basic technique of the approach. Section 5.2.2 presents the precondition removal technique to complement the basic technique. Section 5.2.3 describes the iterative process of applying these techniques.

5.2.1 Basic technique

In the basic technique (Figure 5.2), we run the existing unit test suite on the program that is instrumented by the Daikon front end. The execution produces a data trace file, which contains variable values computed during execution. Then we use Daikon to infer operational abstractions from the data trace file. We extend the Daikon toolset to insert the operational abstractions into the source code as DbC annotations. We feed the resulting annotated code to Jtest, which automatically gener-

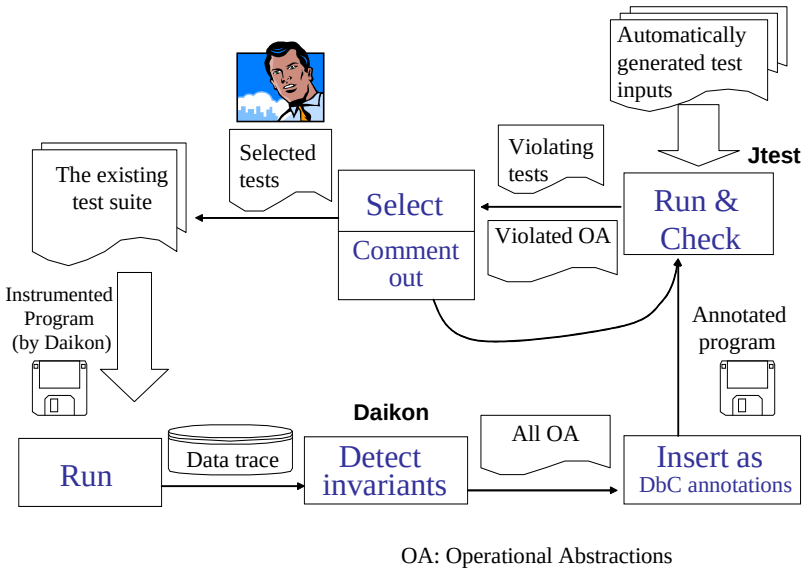

Figure 5.2: An overview of the basic technique

ates and executes new tests. The two symptoms of an operational violation are that an operational abstraction is evaluated to be `false`, or that an exception is thrown while evaluating an operational abstraction. When a certain number of operational violations have occurred before Jtest exhausts its testing repository, Jtest stops generating test inputs and reports operational violations. Jtest exports all the operational violations, including the violating test inputs, to a text file. Given the exported text file, we automatically comment out the violated operational abstractions in the source code. At the same time, we collect the operational violations. Then we invoke Jtest again, which is given the program with reduced operational abstractions. We repeat the preceding procedure iteratively until we cannot find any operational violations. We call these iterations as *inner iterations* to avoid their being confused with the iterations described in Section 5.2.3. The inner iterations mainly comprise the activities of Jtest's test generation and execution, Jtest's violation report, and our violated-operational-abstraction collection and removal. These inner iterations enable Jtest to fully generate violating tests.

Given the collected operational violations, we select the first encountered test for each violated operational abstraction. So when there are multiple tests that violate the same operational abstraction, we select only the first encountered one instead of all of them. Since a selected violating test

might violate multiple operational abstractions, we group together all of the operational abstractions violated by the same test. Then we sort the selected violating tests based on the number of their violated operational abstractions. We put the tests that violate more operational abstractions before those that violate fewer ones. The heuristic behind this is that a test that violates more operational abstractions might be more valuable than a test that violates fewer ones. When programmers cannot afford to inspect all violating tests, they can inspect just the top parts of the prioritized tests.

We finally produce a JUnit [GB03] test class, which contains the sorted list of violating test inputs as well as their violated operational abstractions. We developed a set of integration tools in Perl to fully automate the preceding steps, including invoking Daikon and Jtest, and postprocessing the text file. After running the integration tools, programmers can then execute or inspect the resulting sorted tests to verify the correctness of their executions. Optionally, programmers can add assertions for the test inputs as test oracles for regression testing.

One example of operational violations is shown in Figure 5.3. The example indicates a deficiency of the JAX test suite. The top part of Figure 5.3 shows two relevant tests (JAX Tests 1 and 2) used for inferring the `isMember` method's two violated postconditions (`assertTrue` in the tests is JUnit's built-in assertion method). The postconditions are followed by the violating test input generated by Jtest. In the postconditions, `@post` is used to denote postconditions. The `$pre` keyword is used to refer to the value of an expression immediately before calling the method; the syntax to use `$pre` is `$pre(expressionType, expression)`. The `$result` keyword is used to represent the return value of the method.

The violated postconditions show the following behavior exhibited by the existing tests:

- The `isMember(3)` method is invoked iff its return value is `true`.

- The `isMember(3)` method is invoked iff the `numberOfElements` (after the method invocation) is `1`.

The test input of invoking `isMember(3)` method on an empty stack violates these two ungeneralizable postconditions.

```
JAX Test 1:
  UBStack stack = new UBStack();
  assertTrue(!stack.isMember(2));
JAX Test 2:
  UBStack stack1 = new UBStack();
  UBStack stack2 = new UBStack();
  stack1.push(3);
  assertTrue(stack1.isMember(3));
  stack1.push(2);
  stack1.push(1);//because max is 2, this push cannot put 1 into stack1
  stack2.push(3);
  stack2.push(2);
  //the following assertion makes sure 1 is not in stack1
  assertTrue(stack1.isMember(1) == stack2.isMember(1));

Inferred postconditions for isMember:
  @post: [($pre(int, k) == 3) == ($result == true)]
  @post: [($pre(int, k) == 3) == (this.numberOfElements == 1)]

Violating Jtest-generated test input:
  UBStack THIS = new UBStack ();
  boolean RETVAL = THIS.isMember (3);
```

Figure 5.3: An example of operational violations using the basic technique

5.2.2 Precondition removal technique

In the basic technique, when the existing test suite is deficient, the inferred preconditions might be overconstrained so that Jtest filters out valid test inputs during test generation and execution. However, we often need to exercise the unit under more circumstances than what is constrained by the inferred overconstrained preconditions. To address this, before we feed the annotated code to Jtest, we use a script to automatically remove all inferred preconditions, and we thus enable Jtest to exercise the unit under a broader variety of test inputs. Indeed, removing preconditions can make test generation tools less guided, especially those tools that generate tests mainly based on preconditions [BKM02]. Another issue with this technique is that removing inferred preconditions allows test generation tools to generate invalid test inputs if some values of a parameter type are invalid.

Figure 5.4 shows one example of operational violations and the use of this technique. `@invariant` is used to denote class invariants. The example indicates a deficiency of the basic JUnit test suite, and the violating test exposes the fault detected by the original JAX test suite. The violated post-conditions and invariant show the following behavior exhibited by the existing tests:

- After the invocation of the `pop()` method, the element on top of the stack is equal to the element on the second to top of the stack before the method invocation.
- After the invocation of the `pop()` method, the `numberOfElements` is equal to `0` or `1`.
- In the entries and exits of all the public methods, the `numberOfElements` is equal to `0`, `1`, or `2`.

Since the capacity of the stack is `2`, the inferred behavior seems to be normal and consistent with our expectation. Jtest generates a test that invokes `pop()` on an empty stack. In the exit of the `pop()` method, the `numberOfElements` is equal to `-1`. This value causes the evaluation of the first postcondition to throw an exception, and the evaluation of the second postcondition or the invariant to get the `false` value. By looking into the specifications [SLA02] for `UBStack`, we can know that the implementation does not appropriately handle the case where the `pop()` method is invoked on an empty stack; the specifications specify that the empty stack should maintain the same empty state when the `pop()` method is invoked.

The example in Figure 5.5 shows a deficiency of the JAX test suite, and the violating test exposes another new fault. This fault is not reported in the original experiment [SLA02]. The inferred postcondition states that the method return is equal to `-1` iff the `numberOfElements` is equal to `0`. The code implementer uses `-1` as the error indicator for calling the `top()` method on an empty stack instead of an `topEmptyStack` exception specified by the specifications [SLA02] . According to the specifications, this stack should also accommodate negative integer elements; this operational violation shows that using `-1` as an error indicator makes the `top` method work incorrectly when the `-1` element is put on top of the stack. This is a typical value-sensitive fault and even a full-path-coverage test suite cannot guarantee to expose this fault. The basic technique does not report this violation because of the overconstrained preconditions. The existing tests push only positive integers into the stack, so Daikon infers several preconditions for the `top` method, which prevent

```
Basic JUnit Test 1:
  UBStack stack = new UBStack();
  stack.push(3);
  stack.pop();
Basic JUnit Test 2:
  UBStack stack = new UBStack();
  stack.push(3);
  stack.push(2);
  stack.pop();

Inferred postconditions for pop:
  @post: [( this.elems[this.numberOfElements] ==
        this.elems[$pre(int, this.numberOfElements)-1] )]
  @post: [this.numberOfElements == 0 ||
          this.numberOfElements == 1]
Inferred class invariant for UBStack:
  @invariant: [this.numberOfElements == 0 ||
               this.numberOfElements == 1 ||
               this.numberOfElements == 2]

Violating Jtest-generated test input:
 UBStack THIS = new UBStack ();
 THIS.pop ();
```

Figure 5.4: The first example of operational violations using the precondition removal technique

---

the `-1` element from being on top of the stack. One such precondition is:

```
@pre:  for (int i = 0 ; i <= this.elems.length-1; i++)
           $assert ((this.elems[i] >= 0));
```

where `@pre` is used to denote a precondition and `$assert` is used to denote an assertion statement within the loop body. Both the loop and the assertion statement form the precondition.

5.2.3 Iterations

After we perform the test selection using the techniques in Sections 5.2.1 and 5.2.2, we can further run all the violating tests together with the existing ones to infer new operational abstractions. By doing so, we can automatically remove or weaken the operational abstractions violated by the vi-

```
JAX Test 3:
  UBStack stack = new UBStack();
  stack.push(3);
  stack.push(2);
  stack.pop();
  stack.pop();
  stack.push(3);
  stack.push(2);
  int oldTop = stack.top();
JAX Test 4:
  UBStack stack = new UBStack();
  assertTrue(stack.top() == -1);
JAX Test 5:
  UBStack stack1 = new UBStack();
  UBStack stack2 = new UBStack();
  stack1.push(3);
  assertTrue(stack1.top() == 3);
  stack1.push(2);
  stack1.push(1);
  stack2.push(3);
  stack2.push(2);
  assertTrue(stack1.top() == stack2.top());
  stack1.push(3);
  assertTrue(stack1.top() == 3);

Inferred postcondition for top:
 @post: [($result == -1) == (this.numberOfElements == 0)]

Violating Jtest-generated test input:
 UBStack THIS = new UBStack ();
 THIS.push (-1);
 int RETVAL = THIS.top ();
```

Figure 5.5: The second example of operational violations using the precondition removal technique

---

olating tests. Based on the new operational abstractions, Jtest might generate new violating tests for the weakened or other new operational abstractions. We repeat the process described in Sections 5.2.1 and 5.2.2 until there are no reported operational violations or until the user-specified maximum number of iterations has been reached. We call these iterations as *outer iterations*. Dif-

```
(1st iteration)
Inferred postcondition for isMember:
 @post: [($result == true) == (this.numberOfElements == 1)]

Violating Jtest-generated test input:
 UBStack THIS = new UBStack ();
 THIS.top ();
 THIS.push (2);
 boolean RETVAL = THIS.isMember (1);

(2nd iteration)
Inferred postcondition for isMember:
 @post:[($result == true) $implies (this.numberOfElements == 1)]

Violating Jtest-generated test input:
 UBStack THIS = new UBStack ();
 THIS.push (2);
 THIS.push (0);
 boolean RETVAL = THIS.isMember (0);
```

Figure 5.6: Operational violations during iterations

ferent from the inner iterations described in Section 5.2.1, these outer iterations operate in a larger scale. They mainly comprise the activities of the existing tests' execution, Daikon's operational-abstraction generation, our DbC annotation insertion, the inner iterations, and our test selection and augmentation. We have used a script to automate the outer iterations. In the rest of the chapter, for the sake of brevity, iterations will refer to outer iterations by default.

Figure 5.6 shows two operational violations during the first and second iterations on the JAX test suite. The JAX test suite exhibits that the return of the `isMember()` method is `true` iff the `numberOfElements` after the method execution is equal to `1`. In the first iteration, a violating test shows that if the `numberOfElements` after the method execution is equal to `1`, the return of the `isMember()` method is not necessarily `true` (it can be `false`). After the first iteration, we add this violating test to the existing test suite. In the second iteration, with the augmented test suite, Daikon infers an updated postcondition by weakening the `==` predicate (meaning iff or $\Leftrightarrow$) to the `$implies` predicate (meaning $\Rightarrow$). The updated postcondition shows that if the return of the

`isMember()` method is `true`, the `numberOfElements` after the method execution is equal to `1`. In the second iteration, another violating test shows that if the return of the `isMember()` method is `true`, the `numberOfElements` after the method execution is not necessarily equal to `1` (it can be equal to `2`). After the second iteration, we add this violating test to the existing test suite. In the third iteration, Daikon eliminates this `$implies` predicate since Daikon does not observe any correlation between the return of the `isMember()` method and the `numberOfElements`.

5.3 Evaluation

Testing is used not only for finding bugs but also for increasing our confidence in the code under test. For example, generating and selecting tests for achieving better structural coverage can increase our confidence in the code although they do not find bugs; indeed, these tests can be used as regression tests executed on later versions for detecting regression bugs. Although our approach tries to fill gaps in the existing test suite or to identify its weakness in order to improve its quality, our approach does not intend to be considered as a general approach for generating and selecting tests (based on the current program version) to increase the existing test suite's capability of exposing future arbitrarily introduced bugs (on future program versions) during program maintenance. Therefore, when we designed our experiments for assessing the approach, we did not use mutation testing [BDLS80] to measure the capability of the selected tests in finding arbitrary bugs in general. Instead, we conducted experiments to primarily measure the capability of the selected tests in revealing anomalous behavior on the real code, such as revealing a fault in terms of correctness or a failure in terms of robustness. We do not distinguish these two types of anomalous behavior because in the absence of specifications we often could not distinguish these two cases precisely. For example, the violating tests shown in Figure 5.4 and Figure 5.5 would have been considered as invalid tests for revealing failures if the actual precondition for `pop()` were `(this.numberOfElements > 0)` and the actual precondition for `push(int k)` were `(k >= 0)`; however, these two tests are valid fault-revealing tests based on `UBStack`'s specifications [SLA02]. Indeed, we could try to hand-construct specifications for these programs; however, the code implementation and comments for these programs alone are not sufficient for us to recover the specifications (especially preconditions) easily and we do not have easy access to the program intentions originally residing in code authors' mind.

Note that if a selected test does not expose anomalous behavior, it might still provide value in filling gaps in the existing test suite. However, in the absence of specifications, it would be too subjective in judging these tests in terms of providing value; therefore, we did not perform such a subjective judgement in our experiments.

In particular, the general questions we wish to answer include:

1. Is the number of automatically generated tests large enough for programmers to adopt unit-test selection techniques?

2. Is the number of tests selected by our approach small enough for programmers to inspect affordably?

3. Do the tests selected by our approach have a high probability of exposing anomalous program behavior?

4. Do the operational abstractions guide test generation tools to better generate tests for violating the operational abstractions?

We cannot answer all of these questions easily, so we designed experiments to give an initial sense of the general questions of efficacy of this approach. In the remaining of this section, we first describe the measurements in the experiments. We then present the experiment instrumentation. We finally describe the experimental results and threats to validity.

5.3.1 Measurements

In particular, we collected the following measurements to address these questions directly or indirectly:

- Automatically generated test count in the absence of any operational abstraction (`#AutoT`): We measured the number of tests automatically generated by Jtest alone in the absence of any operational abstraction. We call these tests as *unguided-generated tests*. This measurement is related to the first question.

- Selected test count (`#SelT`): We measured the number of the tests selected by a test selection technique. This measurement is related to the second question, as well as the fourth question.

- Anomaly-revealing selected test count (#ART): We measured the number of anomaly-revealing tests among the selected tests. These anomaly-revealing tests expose anomalous program behavior (related to either faults in terms of correctness or failures in terms of robustness). After all the iterations terminate, we manually inspect the selected tests, violated postconditions, and the source code to determine the anomaly-revealing tests. Although our test selection mechanism described in Section 5.2.1 guarantees that no two selected tests violate the same set of postconditions, multiple anomaly-revealing tests might suggest the same precondition or expose the same fault in different ways. This measurement is related to the third question, as well as the fourth question.

We collected the #AutoT measurement for each subject program. We collected the #SelT and #ART measurements for each combination of the basic/precondition removal techniques, subject programs, and number of iterations. These measurements help answer the first three questions.

To help answer the fourth question, we used Jtest alone to produce unguided-generated tests, then ran the unguided-generated tests, and check them against the operational abstractions (keeping the preconditions) generated from the existing tests. We selected those unguided-generated tests that satisfied preconditions and violated postconditions. We then collected the #SelT and #ART measurements for each subject program, and compared the measurements with the ones for the basic technique.

In addition, we used Jtest alone to produce unguided-generated tests, then ran the unguided-generated tests, and check them against the operational abstractions (removing the preconditions) generated from the existing tests. We selected those unguided-generated tests that violated postconditions. We then collected the #SelT and #ART measurements for each subject program, and compared the measurements with the ones for the precondition removal technique.

5.3.2 Experiment instrumentation

Table 5.1 lists the subject programs that we used in the experiments. Each subject program is a Java class equipped with a manually written unit test suite. The first column shows the names of the subject programs. The second and third columns show the number of public methods, and the number of lines of executable code for each program, respectively. The fourth column shows

Table 5.1: Subject programs used in the experiments.

program	*#pmethod*	*#loc*	*#tests*	`#AutoT`	`#ExT`
`UB-Stack(JUnit)`	11	47	8	96	1
`UB-Stack(JAX)`	11	47	15	96	1
`RatPoly-1`	13	161	24	223	1
`RatPoly-2`	13	191	24	227	1
`RatPolyStack-1`	13	48	11	128	4
`RatPolyStack-2`	12	40	11	90	3
`BinaryHeap`	10	31	14	166	2
`BinarySearchTree`	16	50	15	147	0
`DisjSets`	4	11	3	24	4
`QueueAr`	7	27	11	120	1
`StackAr`	8	20	16	133	1
`StackLi`	9	21	16	99	0

the number of test cases in the test suite of each program. The last two columns present some measurement results that we shall describe in Section 5.3.3.

Among these subjects, `UB-Stack(JUnit)` and `UB-Stack(JAX)` are the example (Section 5.1) with the basic JUnit test suite and the JAX test suite (with one failing test removed), respectively [SLA02]. `RatPoly-1`/`RatPoly-2` and `RatPolyStack-1`/`RatPolyStack-2` are the student solutions to two assignments in a programming course at MIT. These selected solutions passed all the unit tests provided by instructors. The rest of the subjects come from a data structures textbook [Wei99]. Daikon group members developed unit tests for 10 data structure classes in the textbook. Most of these unit tests use random inputs to extensively exercise the programs. We applied our approach on these classes, and five classes (the last five at the end of Table 5.1) have at least one operational violation.

In the experiments, we used Daikon and Jtest to implement our approach. We developed a set of Perl scripts to integrate these two tools. In Jtest's configuration for the experiments, we set the length of calling sequence as two. We used Daikon's default configuration for the generation of operational abstractions except that we turned on the inference of conditional invariants. In particular,

we first ran Jtest on each subject program to collect the #AutoT measurement in the absence of any operational abstraction. We exported the unguided-generated tests for each program to a JUnit test class. Then for each program, we conducted the experiment using the basic technique, and repeated it until we reached the third iteration or until no operational violations were reported for the operational abstractions generated from the previous iteration. At the end of each iteration, we collected the #SelT and #ART measurements. We performed a similar procedure for the precondition removal technique.

5.3.3 Experimental results

The fifth column of Table 5.1 shows the #AutoT results. From the results, we observed that except for the especially small DisjSets program, Jtest automatically generated nearly 100 or more tests. We also tried setting the length of the calling sequence to three, which caused Jtest to produce thousands of tests for the programs. This shows that we need test selection techniques since it is not practical to manually check the outputs of all these automatically generated tests.

The last column (#ExT) of Table 5.1 shows the number of the automatically generated tests that cause uncaught runtime exceptions. In the experiments, since all the test selection methods under comparison additionally select this type of tests, the #SelT and #ART measurements do not count them for the sake of better comparison.

Table 5.2 and Table 5.3 show the #SelT and #ART measurements for the basic technique and the precondition removal technique, respectively. In either table, the *iteration 1*, *iteration 2*, and *iteration 3* columns show the results for three iterations. In Table 5.2, the *unguided* column shows the results for selecting unguided-generated tests that satisfy preconditions and violate postconditions. In Table 5.3, the *unguided* column shows the results for selecting unguided-generated tests that violate postconditions (no matter whether they satisfy preconditions). In either table, for those #SelT with the value of zero, their entries and their associated #ART entries are left blank. The bottom two rows of either table show the median and average percentages of #ART among #SelT. In the calculation of the median or average percentage, the entries with a #SelT value of zero are not included.

The numbers of tests selected by both techniques vary across different programs but on average

Table 5.2: The numbers of selected tests and anomaly-revealing selected tests using the basic technique for each iteration and the unguided-generated tests

program	*iteration 1*		*iteration 2*		*iteration 3*		*unguided*	
	#SelT	#ART	#SelT	#ART	#SelT	#ART	#SelT	#ART
UB-Stack(JUnit)	1	0	2	0				
UB-Stack(JAX)	3	0						
RatPoly-1	2	2						
RatPoly-2	1	1	1	1				
RatPolyStack-1								
RatPolyStack-2	1	0						
BinaryHeap	3	2	1	0			2	2
BinarySearchTree								
DisjSets	1	1					1	1
QueueAr	6	1					2	1
StackAr	5	1	1	0			1	1
StackLi								
median(#ART/#SelT)	20%		0%		0%		100%	
average(#ART/#SelT)	45%		25%		0%		88%	

their numbers are not large, so their executions and outputs could be verified with affordable human effort. The basic technique selects fewer tests than the precondition removal technique. This is consistent with our hypothesis that the basic technique might overconstrain test generation tools. We observed that the number of tests selected by either technique is higher than the number of tests selected by checking unguided-generated tests against operational abstractions. This indicates that operational abstractions guide Jtest to better generate tests to violate them. Specifically, the precondition removal technique gives more guidance to Jtest for generating anomaly-revealing tests than the basic technique. There are only two subjects for which the basic technique generates anomaly-revealing tests but Jtest alone does not generate any (shown in Table 5.2); however, the precondition removal technique generates more anomaly-revealing tests than Jtest alone for most subjects (shown in Table 5.3).

We observed that, in the experiments, the selected tests by either technique have a high probability of exposing anomalous program behavior. In the absence of specifications, we suspect that many

Table 5.3: The numbers of selected tests and anomaly-revealing selected tests using the precondition removal technique for each iteration and the unguided-generated tests

program	*iteration 1*		*iteration 2*		*iteration 3*		*unguided*	
	#SelT	#ART	#SelT	#ART	#SelT	#ART	#SelT	#ART
UB-Stack(JUnit)	15	5	6	1	1	0	4	1
UB-Stack(JAX)	25	9	4	0			3	1
RatPoly-1	1	1						
RatPoly-2	1	1					1	1
RatPolyStack-1	12	8	5	2	1	0		
RatPolyStack-2	10	7	2	0				
BinaryHeap	8	6	8	6	6	0	4	3
BinarySearchTree	3	3					1	1
DisjSets	2	2					1	1
QueueAr	11	1	4	1			4	1
StackAr	9	1	1	0			1	1
StackLi	2	0					1	0
median(#ART/#SelT)	68%		17%		0%		75%	
average(#ART/#SelT)	58%		22%		0%		62%	

of these anomaly-revealing tests are failure-revealing test inputs; programmers can add preconditions, condition-checking code, or just pay attention to the undesirable behavior when the code's implicit assumptions are not written down.

We describe a concrete case for operational violations in the experiments as follows. `RatPoly-1` and `RatPoly-2` are two student solutions to an assignment of implementing `RatPoly`, which represents an immutable single-variate polynomial expression, such as "0", "$x - 10$", and "$x^3 - 2 * x^2 + 53 * x + 3$". In `RatPoly`'s class interface, there is a method `div` for `RatPoly`'s division operation, which invokes another method `degree`; `degree` returns the largest exponent with a non-zero coefficient, or 0 if the `RatPoly` is "0". After we ran with Daikon the instructor-provided test suite on both `RatPoly-1` and `RatPoly-2`, we got the same DbC annotations for both student solutions. The precondition removal technique selects one violating test for each student solution. The selected violating test for `RatPoly-1` is different from the one for `RatPoly-2`; this result shows that Jtest takes the code implementation into account when generating tests to violate the given DbC

```
Inferred postcondition for degree:
  $result >= 0

Violating Jtest-generated test input (for RatPoly-1):
   RatPoly t0 = new RatPoly(-1, -1);//represents -1*x^-1
   RatPoly THIS = new RatPoly (-1, 0);//represents -1*x^0
   RatPoly RETVAL = THIS.div (t0);//represents (-1*x^0)/(-1*x^-1)

Violating Jtest-generated test input (for RatPoly-2):
   RatPoly t0 = new RatPoly(1, 0);//represents 1*x^0
   RatPoly THIS = new RatPoly (1, -1);//represents 1*x^-1
   RatPoly RETVAL = THIS.div (t0);//represents (1*x^-1)/(1*x^0)
```

Figure 5.7: Operational violations for `RatPoly-1`/`RatPoly-2`

---

annotations. The selected test for `RatPoly-1` makes the program infinitely loop until a Java out-of-memory error occurs and the selected test for `RatPoly-2` runs normally with termination and without throwing exceptions. These tests are not generated by Jtest alone without being guided with operational abstractions. After inspecting the code and its comments, we found that these selected tests are invalid, because there is a precondition `e >= 0` for `RatPoly(int c, int e)`. This case shows that the operational abstraction approach can help generate test inputs to crash a program and then programmers can improve their code's robustness when specifications are absent.

We observed that although those non-anomaly-revealing selected tests do not expose any fault, most of them represent some special class of inputs, and thus may be valuable if selected for regression testing. We observed, in the experiments, that a couple of iterations are good enough in our approach. Jtest's test generation and execution time dominates the running time of applying our approach. Most subjects took several minutes, but the `BinaryHeap` and `RatPolyStack` programs took on the order of 10 to 20 minutes. We expect that the execution time can be optimized if future versions of Jtest can better support the resumption of test generation and execution after we comment out the violated operational abstractions.

5.3.4 Threats to validity

The threats to external validity primarily include the degree to which integrated third-party tools, the subject programs, and test cases are representative of true practice. These threats could be reduced by more experiments on wider types of subjects and third-party tools. Parasoft Jtest 4.5 is one of the testing tools popularly used in industry and the only specification-based test generation tool available to us at the moment. Daikon is the only publicly available tool for generating operational abstractions. Daikon's scalability has recently been tackled by using incremental algorithms for invariant detection [PE04]. In our approach, we use Daikon to infer invariants based on only manual tests in addition to selected violating tests; the size of these tests is often small. However, Jtest 4.5 is not designed for being used in an iterative way; if some operational abstractions can be violated, we observed that the number of inner iterations can be more than a dozen and the elapsed time could be longer than five minutes for some subjects. We expect that the scalability of Jtest in our setting could be addressed by enhancing it to support incremental test generation when DbC annotations are being changed. Furthermore, the elapsed time for Jtest's test generation can be reduced by enhancing it to avoid generating redundant tests (described in Chapter 4). Alternatively we can use other specification-based tools with more efficient mechanisms for test generation, such as Korat [BKM02].

We mainly used data structures as our subject programs and the programs are relatively small (the scalability of Jtest 4.5 poses difficulties for us to try large subjects, but note that this is not the inherent limitation of our approach but the limitation of one particular implementation of our approach). Although data structures are better suited to the use of invariant detection and design-by-contract specifications, Daikon has been used on wider types of programs [Dai04]. The success of our approach on wider types of programs also depends on the underlying testing tool's capability of generating test inputs to violate specifications if there exist violating test inputs. We expect that the potential of our approach for wider types of programs could be further improved if we use specification-based testing tools with more powerful test generation capability, such as Korat [BKM02], CnC [CS05], and our Symstra tool presented in Chapter 4.

The main threats to internal validity include instrumentation effects that can bias our results. Faults in our integration scripts, Jtest, or Daikon might cause such effects. To reduce these threats,

we manually inspected the intermediate results of most program subjects. The main threats to construct validity include the uses of those measurements in our experiments to assess our approach. We measured the number of anomaly-revealing tests to evaluate the value of selected tests. In future work, we plan to measure some other possible attributes of the selected tests.

5.4 Conclusion

Selecting automatically generated test inputs to check correctness and augment the existing unit test suite is an important step in unit testing. Inferred operational abstractions act as a summary of the existing test execution history. These operational abstractions can guide test generation tools to better produce test inputs to violate the abstractions. We have developed the operational violation approach for selecting generated tests that violate operational abstractions; these selected violating tests are good candidates for inspection, since they exercise new program features that are not covered by the existing tests. We have conducted experiments on applying the approach on a set of data structures. Our experimental results have shown that the size of the selected tests is reasonably small for inspection, the selected tests generally expose new interesting behavior filling the gaps not covered by the existing test suite, and the selected tests have a high probability of exposing anomalous program behavior (either faults or failures) in the code.

Our approach shows a feedback loop between behavior inference and test generation. The feedback loop starts with existing tests (constructed manually or automatically) or some existing program runs. After running the existing tests, a behavior inference tool can infer program behavior exercised by the existing tests. The inferred behavior can be exploited by a test-generation tool in guiding its test generation, which generates new tests to exercise new behavior. Some generated tests may violate the inferred proprieties (the form of the inferred behavior) and these violating tests are selected for inspection. Furthermore, these selected tests are added to the existing tests. The existing tests augmented by the new selected tests can be used by the behavior inference tool to infer behavior that is closer to what shall be described by a specification (if it is manually constructed) than the behavior inferred from the original existing tests. The new inferred behavior can be further used to guide test generation in the subsequent iteration. Iterations terminate until a user-defined maximum iteration number has been reached or no new behavior has been inferred from new tests.

This feedback loop provides a means to producing better tests and better approximated specifications automatically and incrementally. The feedback loop not only allows us to gain benefits of specification-based testing in the absence of specifications, but also tackles one issue of dynamic behavior inference: the quality of the analysis results (inferred behavior) heavily depends on the quality of the executed tests.

Chapter 6

TEST ABSTRACTION FOR INSPECTION

Automatic test-generation tools can generate a large number of tests for a class. Without a prior specifications, developers usually rely on uncaught exceptions or inspect the execution of generated tests to determine program correctness. However, relying on only uncaught exceptions for catching bugs is limited and inspecting the execution of a large number of generated tests is impractical. The operational violation approach presented in Chapter 5 selects a subset of generated tests for inspection; these selected tests exhibit new behavior that has not been exercised by the existing tests. In this chapter, we present the *observer abstraction* approach that abstracts and summarizes the object-state-transition information collected from the execution of generated tests. Instead of inspecting the execution of individual tests, developers can inspect the summarized object-state-transition information for various purposes. For example, developers can inspect the information to determine whether the class under test exhibits expected behavior. Developers can also inspect the information to investigate causes of the failures exhibited by uncaught exceptions. Developers can inspect the information for achieving better understanding of the class under test or even the tests themselves.

From the execution of tests, we can construct an object state machine (OSM): a state in an OSM represents the state that an object is in at runtime. A transition in an OSM represents method calls invoked through the class interface transiting the object from one state to another. States in an OSM can be represented by using concrete or abstract representation. The *concrete-state representation* of an object, in short as *concrete object state*, is characterized by the values of all the fields transitively reachable from the object (described in Section 3.2.2 of Chapter 3). A concrete OSM is an OSM whose states are concrete object states. Because a concrete OSM is often too complicated to be useful for understanding, we extract an abstract OSM that contains abstract states instead of concrete states. An *abstract state* of an object is defined by an *abstraction function* [LG00]; the abstraction function maps each concrete state to an abstract state. Our observer abstraction approach defines

abstraction functions automatically by using an *observer*, which is a public method with a non-void return.[1] In particular, the observer abstraction approach abstracts a concrete object state exercised by tests based on the return values of a set of observers that are invoked on the concrete object state. An *observer abstraction* is an OSM whose states are represented by abstract representations that are produced based on observers. We have implemented a tool, called Obstra, for the observer abstraction approach. Given a Java class and its initial unit test (either manually constructed or auotmatically generated), Obstra identifies concrete object states exercised by the tests and generates new tests to augment these initial tests. Based on the return values of a set of observers, Obstra maps each concrete object state to an abstract state and constructs an OSM.

The next section describes the example that we use to illustrate our approach. Section 6.2 presents the observer abstraction approach. Section 6.3 describes our experiences of applying the approach on several data structures and then Section 6.4 concludes.

6.1 Example

We use a binary search tree implementation as a running example to illustrate our observer abstraction approach. Figure 6.1 shows the relevant parts of the code. The class has 246 non-comment, non-blank lines of code and its interface includes eight public methods (five of them are observers), some of which are a constructor (denoted as `[init]()`), `boolean contains(MyInput info)`, `void add(MyInput info)`, and `boolean remove (MyInput info)`. The `MyInput` argument type contains an integer field `v`, which is set through the class constructor. `MyInput` implements the `Comparable` interface and two `MyInput` are compared based on the values of their `v` fields. Parasoft Jtest 4.5 [Par03] generates 277 tests for the class.

6.2 Observer Abstraction Approach

We first discuss the test argumentation technique that enables the dynamic extraction of observer abstractions (Section 6.2.1). We next describe object state machines, being the representations of

[1] We follow the definition by Henkel and Diwan [HD03]. The definition differs from the more common definition that limits an observer to methods that do not change any state. We have found that state-modifying observers also provide value in our approach and state modification does not harm our approach.

```
class BST implements Set {
    Node root;
    static class Node {
        MyInput info;
        Node left;
        Node right;
    }
    public void add(MyInput info) {
        if (root == null) { root = new Node(); root.info = info;  }
        else {
            Node t = root;
            while (true) {
                if (t.info.compareTo(info) < 0) { ... }
                else if (t.value.compareTo(info) > 0) { ... }
                else { /* no duplicates*/   return;  }
            }
        }
    }
    public boolean remove(MyInput info) {
        Node parent = null;   Node current = root;
        while (current != null) {
            if (info.compareTo(current.info) < 0) { ... }
            else if (info.compareTo(current.info) > 0) { ... }
            else { break; }
        }
        if (current == null) return false;
        ...
        return true;
    }
    public boolean contains(MyInput info) { ... }
    ...
}
```

Figure 6.1: A set implemented as a binary search tree

---

observer abstractions (Section 6.2.2). We then define observer abstractions and illustrate dynamic extraction of them (Section 6.2.3).

6.2.1 Test Augmentation

We use the WholeState technique to represent the concrete state of an object (Section 3.2.2 of Chapter 3). The technique represents the *concrete object state* of an object as the heap rooted from the object; the rooted heap is further linearized to a sequence of the values of the fields transitively reachable from the object. Two concrete object states are equivalent iff their rooted heaps are isomorphic. A set of *nonequivalent concrete object states* contain concrete object states any two of which are not equivalent. A *method-argument state* is characterized by a method and the values for the method arguments (Section 4.2 of Chapter 4). Two method-argument states are equivalent iff their methods are the same and the heaps rooted from their method arguments are isomorphic. A set of *nonequivalent method-argument states* contain method-argument states any two of which are not equivalent.

After we execute an initial test suite, the WholeState technique identifies all nonequivalent object states and nonequivalent method-argument states that were exercised by the test suite. We then apply the test augmentation technique that generates new tests to exercise each possible combination of nonequivalent object states and nonequivalent non-constructor-method-argument states. A combination of a receiver-object state and a method-argument state forms a method invocation. We augment the initial test suite because the test suite might not invoke each observer on all nonequivalent object states; invoking observers on a concrete object state is necessary for us to know the abstract state that encloses the concrete object state. The augmented test suite guarantees the invocation of each nonequivalent non-constructor method-argument state on each nonequivalent object state at least once. In addition, the observer abstractions extracted from the augmented test suite can better help developers to inspect object-state-transition behavior. The complexity of the test augmentation algorithm is $O(|CS| \times |MC|)$, where CS is the set of the nonequivalent concrete states exercised by the initial test suite T for the class under test and MC is the set of the nonequivalent method calls exercised by T.

6.2.2 Object State Machine

We define an object state machine for a class:[2]

Definition 16. *An* object state machine *(OSM) M of a class c is a sextuple M = (I, O, S, δ, λ, $INIT$) where I, O, and S are nonempty sets of method calls in c's interface, returns of these method calls, and states of c's objects, respectively. $INIT \in S$ is the initial state that the machine is in before calling any constructor method of c. $\delta : S \times I \rightarrow P(S)$ is the state transition function and $\lambda : S \times I \rightarrow P(O)$ is the output function where $P(S)$ and $P(O)$ are the power sets of S and O, respectively. When the machine is in a current state s and receives a method call i from I, it moves to one of the next states specified by $\delta(s, i)$ and produces one of the method returns given by $\lambda(s, i)$.*

In the definition, a *method call* is characterized by a method-argument state (a method and the arguments used to invoke the method), not including the receiver-object state. A method call together with a receiver-object state affects the behavior of a method invocation. When a method call in a class interface is invoked on a receiver-object state, an uncaught exception might be thrown. To represent the state where an object is in after an exception-throwing method call, we introduce a special type of states in an OSM: *exception states*. After a method call on a receiver-object state throws an uncaught exception, the receiver object is in an exception state represented by the type name of the exception. The exception-throwing method call transits the object from the object state before the method call to the exception state.

6.2.3 Observer Abstractions

The object states in an OSM can be concrete or abstract. The observer abstraction approach automatically constructs abstraction functions to map a concrete state to an abstract state. These abstraction functions are defined based on observers. We first define an observer following previous work on specifying algebraic specifications for a class [HD03]:

Definition 17. *An* observer *of a class c is a method ob in c's interface such that the return type of ob is not void.*

[2]The definition is adapted from the definition of finite state machine [LY96]; however, an object state machine is not necessarily finite.

For example, `BST`'s observers include `boolean contains(MyInput info)` and `boolean remove (MyInput info)` but `[init]()` and `void add(MyInput info)` are not observers.

An observer call is a method call whose method is an observer. Given a class c and a set of observer calls $OB = \{ob_1, ob_2, ..., ob_n\}$ of c, the observer abstraction approach constructs an abstraction of c with respect to OB. In particular, a concrete state cs is mapped to an abstract state as defined by n values $OBR = \{obr_1, obr_2, ..., obr_n\}$, where each value obr_i represents the return value of observer call ob_i invoked on cs.

Definition 18. *Given a class c and a set of observer calls $OB = \{ob_1, ob_2, ..., ob_n\}$ of c, an* observer abstraction *with respect to OB is an OSM M of c such that the states in M are abstract states defined by OB.*

For example, consider one of `BST`'s observer `contains(MyInput info)`. Jtest generates tests that exercise two observer calls for `contain`: `contains(a0.v:7;)` and `contains(a0:null;)`, where `a`i represents the $(i+1)$th argument and `a`i`.v` represents the `v` field of the $(i+1)$th argument. Argument values are specified following their argument names separated by "`:`" and different arguments are separated by "`;`." Now consider a `BST` object's concrete state cs produced by invoking `BST`'s constructor. Because invoking `contains(a0.v:7;)` or `contains(a0:null;)` on cs returns `false`, the abstract state as for cs is represented by {`false`, `false`}.

Figure 6.2 shows the observer abstraction of `BST` with respect to the two `contains` observer calls and augmented Jtest-generated tests. In the figure, nodes represent abstract states and edges represent state transitions (method calls). The top state in the figure is marked with `INIT`, indicating the object state before invoking a constructor. The second-to-top state is marked with two observer calls and their `false` return values. This abstract state encloses those concrete states such that when we invoke these two observer calls on those concrete states, their return values are `false`. In the central state, the observer calls throw uncaught exceptions and we put the exception-type name `NullPointerException` in the positions of their return values. The bottom state is an exception state, which is marked with the exception-type name `NullPointerException`. An object is in such a state after a method call on the object throws the `NullPointerException`. In the next section, we shall describe transitions in observer abstractions while we present the technique for extracting observer abstractions.

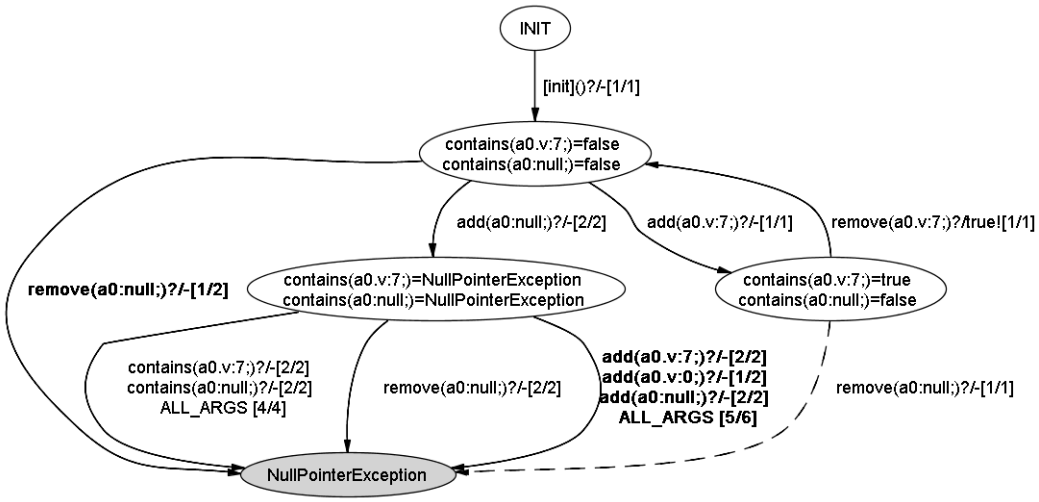

Figure 6.2: `contains` observer abstraction of BST

An OSM can be deterministic or nondeterministic. In a nondeterministic OSM, nondeterministic transitions can offer insights into some irregular object behavior (Section 6.3 shows some examples of exploring nondeterministic transitions). To help characterize nondeterministic transitions, we have defined two numbers in a dynamically extracted OSM: transition counts and emission counts. Assume a transition t transits state S_{start} to S_{end}, the *transition count* associated with t is the number of concrete states enclosed in S_{start} that are transited to S_{end} by t. Assume m is the method call associated with t, the *emission count* associated with S_{start} and m is the number of concrete states enclosed in S_{start} and being at entries of m (but not necessarily being transited to S_{end}). If the transition count of a transition is equal to the associated emission count, the transition is deterministic and nondeterministic otherwise.

Each transition from a starting abstract state to an ending abstract state is marked with method calls, their return values, and some counts. For example, the Jtest-generated test suite for `BST` includes two tests:

```
public class BSTTest extends TestCase {
   public void test1() {
       BST b1 = new BST();
       MyInput m1 = new MyInput(0);
       b1.add(m1);
       b1.remove(null);
```

```
    }

    public void test2() {
        BST b2 = new BST();
        b2.remove(null);
    }
    ...
}
```

The execution of `b1.remove(null)` in `test1` does not throw any exception. Both before and after invoking `b1.remove(null)` in `test1`, if we invoke the two observer calls, their return values are `false`; therefore, there is a state-preserving transition on the second-to-top state. (To present a succinct view, by default we do not show state-preserving transitions.) The execution of `b1.remove(null)` in `test1` throws a `NullPointerException`. If we invoke the two observer calls before invoking `b1.remove(null)` in `test2`, their return values are `false`; therefore, given the method execution of `b1.remove(null)` in `test2`, we extract the transition from the second-to-top state to the bottom state and the transition is marked with `remove(a0:null;)?/−`. In the mark of a transition, when return values are `void` or method calls throw uncaught exceptions, we put "−" in the position of their return values. We put "?" after the method calls and "`!`" after return values if return values are not "−." We also attach two numbers for each transition in the form of `[N/M]`, where `N` is the transition count and `M` is the emission count. If these two numbers are equal, the transition is deterministic, and is nondeterministic otherwise. Because there are two different transitions from the second-to-top state with the same method call `remove(a0:null;)` (one transition is state-preserving being extracted from `test1`), the transition `remove(a0:null;)` from the second-to-top state to the bottom state is nondeterministic, being attached with `[1/2]`. We display thicker edges and bold-font texts for nondeterministic transitions so that developers can easily identify them based on visual effect.

6.2.4 Dynamic Extraction of Observer Abstractions

We dynamically extract observer abstractions of a class from unit-test executions. The number of the concrete states exercised by an augmented test suite is finite and the execution of the test suite is assumed to terminate; therefore, the dynamically extracted observer abstractions are also finite.

Given an initial test suite T for a class c, we first identify the nonequivalent concrete states CS and method-argument states MC exercised by T. We then augment T with new tests to exercise CS with MC exhaustively, producing an augmented test suite T'. We have described these steps in Section 6.2.1. T' exercises each nonequivalent concrete state in CS with each method-argument state in MC; therefore, each nonequivalent observer call in MC is guaranteed to be invoked on each nonequivalent concrete state in CS at least once. We then collect the return values of observer calls in MC for each nonequivalent concrete state in CS. We use this test-generation mechanism to collect return values of observers, instead of inserting observer method calls before and after any call site to the c class in T, because the latter does not work for state-modifying observers, which change the functional behavior of T.

Given an augmented test suite T' and a set of observer calls $OB = \{ob_1, ob_2, ..., ob_n\}$, we go through the following steps to produce an observer abstraction M in the form of OSM. Initially M is empty. During the execution of T', we collect the following tuple for each method execution in c's interface: $(cs_{entry}, m, mr, cs_{exit})$, where cs_{entry}, m, mr, and cs_{exit} are the concrete object state at the method entry, method call, return value, and concrete object state at the method exit, respectively. If m's return type is void, we assign "$-$" to mr. If m's execution throws an uncaught exception, we also assign "$-$" to mr and assign the name of the exception type to cs_{exit}, called an *exception state*. The concrete object state at a constructor's entry is $INIT$, called an *initial state*.

After the test execution terminates, we iterate on each distinct tuple $(cs_{entry}, m, mr, cs_{exit})$ to produce a new tuple $(as_{entry}, m, mr, as_{exit})$, where as_{entry} and as_{exit} are the abstract states mapped from cs_{entry} and cs_{exit} based on OB, respectively. If cs_{exit} is an exception state, its mapped abstract state is the same as cs_{exit}, whose value is the name of the thrown-exception type. If cs_{entry} is an initial state, its mapped abstract state is still $INIT$. If cs_{exit} is not exercised by the initial tests before test augmentation but exercised by new tests, we map cs_{exit} to a special abstract state denoted as N/A, because we have not invoked OB on cs_{exit} yet and do not have a known abstract state for cs_{exit}.

After we produce $(as_{entry}, m, mr, as_{exit})$ from $(cs_{entry}, m, mr, cs_{exit})$, we then add as_{entry} and as_{exit} to M as states, and put a transition from as_{entry} to as_{exit} in M. The transition is denoted by a triple $(as_{entry}, m?/mr!, as_{exit})$. If as_{entry}, as_{exit}, or $(as_{entry}, m?/mr!, as_{exit})$ is already present in M, we do not add it. We also increase the transition count for $(as_{entry}, m?/mr!, as_{exit})$,

denoted as $C_{(as_{entry},m?/mr!,as_{exit})}$, which is initialized to one when $(as_{entry}, m?/mr!, as_{exit})$ is added to M at the first time. We also increase the emission count for as_{entry} and m, denoted as $C_{(as_{entry},m)}$. After we finish processing all distinct tuples $(cs_{entry}, m, mr, cs_{exit})$, we postfix the label of each transition $(as_{entry}, m?/mr!, as_{exit})$ with [$C_{(as_{entry},m?/mr!,as_{exit})}$/$C_{(as_{entry},m)}$]. The complexity of the extraction algorithm for an observer abstraction is $O(|CS| \times |OB|)$, where CS is the set of the nonequivalent concrete states exercised by an initial test suite T and OB is the given set of observers.

To present a succinct view, we do not display N/A states and the transitions leading to N/A states. In addition, we combine multiple transitions that have the same starting and ending abstract states, and whose method calls have the same method names and signatures. When we combine multiple transitions, we calculate the transition count and emission count of the combined transitions and show them in the bottom line of the transition label. When a combined transition contains all nonequivalent method calls of the same method name and signature, we add ALL_ARGS in the bottom line of the transition label. For example, in Figure 6.2, the `contains` edge from the central state to the bottom state is labeled with ALL_ARGS, because the `contains` edge comprises `contains(a0.v:7;)` and `contains(a0:null;)`, which are the only ones for `contains` exercised by the initial test suite.

When a transition contains only method calls that are exercised by new generated tests but not exercised by the initial tests, we display a dotted edge for the transition. For example, in Figure 6.2, there is a dotted edge from the right-most state to the bottom state because the method call for the edge is invoked in the augmented test suite but not in the initial test suite.

To focus on understanding uncaught exceptions, we create a special *exception observer* and construct an observer abstraction based on it. Figure 6.3 shows the exception-observer abstraction of `BST` extracted from the augmented Jtest-generated tests. The exception observer maps the concrete states that are not $INIT$ or exception states to an abstract state called `NORMAL`. The mapped abstract state of an initial state is still $INIT$ and the mapped abstract state of an exception state is still the same as the exception-type name.

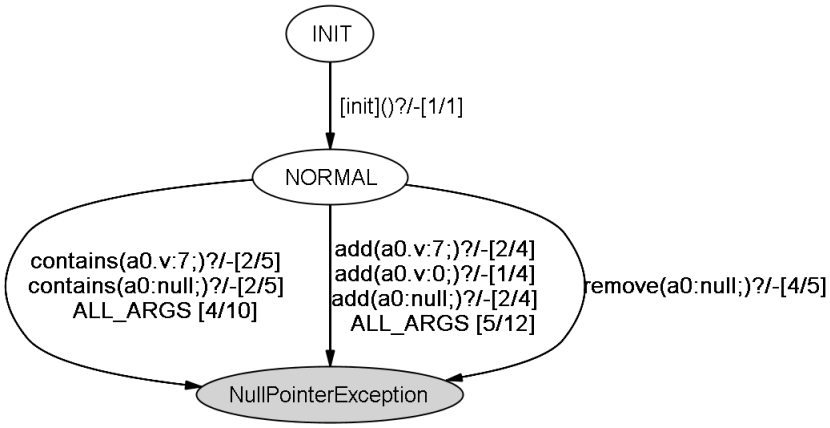

Figure 6.3: `exception` observer abstraction of BST

6.3 Evaluation

We have used Obstra to extract observer abstractions from a variety of programs, most of which were used to evaluate our work in the preceding chapters. Many of these programs manipulate nontrivial data structures. In this section, we illustrate how we applied Obstra on two complex data structures and their automatically generated tests. We applied Obstra on these examples on a MS Windows machine with a Pentium IV 2.8 GHz processor using Sun's Java 2 SDK 1.4.2 JVM with 512 Mb allocated memory.

6.3.1 Binary Search Tree Example

We have described `BST` in Section 6.1 and two of its extracted observer abstractions in Figure 6.2 and 6.3. Jtest generates 277 tests for BST. These tests exercise five nonequivalent concrete object states in addition to the initial state and one exception state, 12 nonequivalent non-constructor method calls in addition to one constructor call, and 33 nonequivalent method executions. Obstra augments the test suite to exercise 61 nonequivalent method executions. The elapsed real time for test augmentation and abstraction extraction is 0.4 and 4.9 seconds, respectively.

Figure 6.3 shows that `NullPointerException` is thrown by three nondeterministic transitions. During test inspection, we want to know under what conditions the exception is thrown. If the exception is thrown because of illegal inputs, we can add necessary preconditions to guard

against the illegal inputs. Alternatively, we can perform defensive programming: we can add input checking at method entries and throw more informative exceptions if the checking fails. However, we do not want to over-constrain preconditions, which would prevent legal inputs from being processed. For example, after inspecting the exception OSM in Figure 6.3, we should not consider that illegal arguments include all arguments for `add`, the `null` argument for `remove`, or all arguments for `contains`, although doing so indeed prevents the exceptions from being thrown. After we inspected the `contains` OSM in Figure 6.2, we gained more information about the exceptions and found that calling `add(a0:null;)` after calling the constructor leads to an undesirable state: calling `contains` on this state deterministically throws the exception. In addition, calling `remove(a0:null;)` also deterministically throws the exception and calling `add` throws the exception with a high probability of 5/6. Therefore, we had more confidence in considering `null` as an illegal argument for `add` and preventing it from being processed. After we prevented `add(a0:null;)`, two `remove(a0:null;)` transitions still throw the exception: one is deterministic and the other is with 1/2 probability. We then considered `null` as an illegal argument for `remove` and prevented it from being processed. We did not need to impose any restriction on the argument of `contains`. Note that this process of understanding the program behavior does not need the access to the source code.

We found that there are three different arguments for `add` but only two different arguments for `contains`, although these two methods have the same signatures. We could add a method call of `contain(a0.v:0;)` to the Jtest-generated test suite; therefore, we could have three observer calls for the `contains` OSM in Figure 6.2. In the new OSM, the second-to-top state includes one more observer call `contains(a0.v:0)=false` and the nondeterministic transition of `remove(a0:null;) ?/-[1/2]` from the second-to-top state to the bottom state is turned into a deterministic transition `remove(a0:null;)?/-[1/1]`. In general, when we add new tests to a test suite and these new tests exercise new observer calls in an OSM, the states in the OSM can be refined, thus possibly turning some nondeterministic transitions into deterministic ones. On the other hand, adding new tests can possibly turn some deterministic transitions into nondeterministic ones.

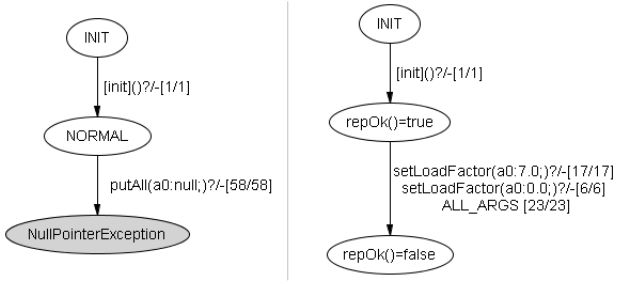

Figure 6.4: `exception` observer abstraction and `repOk` observer abstraction of `HashMap`

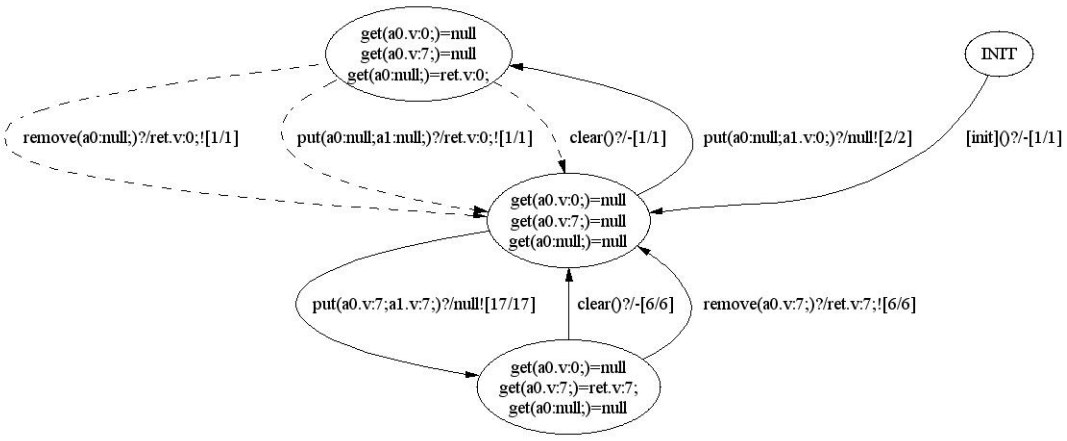

Figure 6.5: `get` observer abstraction of `HashMap`

6.3.2 Hash Map Example

A `HashMap` class was given in `java.util.HashMap` from the standard Java libraries [SM03]. A `repOK` and some helper methods were added to this class for evaluating Korat [BKM02]. The class has 597 non-comment, non-blank lines of code and its interface includes 19 public methods (13 observers), some of which are `[init]()`, `void setLoadFactor(float f)`, `void putAll(Map t)`, `Object remove(MyInput key)`, `Object put(MyInput key, MyInput value)`, and `void clear()`. Jtest generates 5186 tests for `HashMap`. These tests exercise 58 nonequivalent concrete object states in addition to the initial state and one exception state, 29 nonequivalent non-constructor method calls in addition to one constructor call, and 416 nonequivalent method executions. Obstra augments the test suite to exercise 1683 nonequivalent method executions. The elapsed

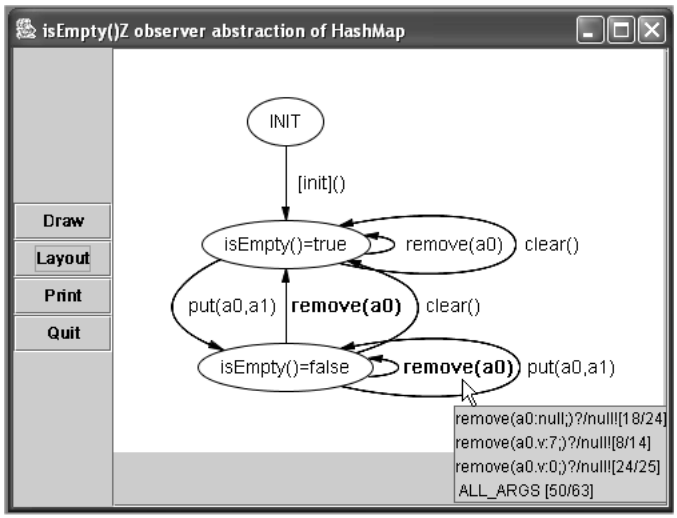

Figure 6.6: `isEmpty` observer abstraction of `HashMap` (screen snapshot)

real time for test augmentation and abstraction extraction is 10 and 15 seconds, respectively.

We found that the exception OSM of `HashMap` contains one deterministic transition, which is `putAll(a0:null;)` from `NORMAL` to `NullPointerException`, as is shown in the left part of Figure 6.4. Therefore, we considered `null` as an illegal argument for `putAll`. We checked the Java API documentation for `HashMap` [SM03] and the documentation states that `putAll` throws `NullPointerException` if the specified map is `null`. This description confirmed our judgment. In other observer abstractions, to provide a more succinct view, by default Obstra does not display any deterministic transitions leading to an exception state in the exception OSM, because the information conveyed by these transitions has been reflected in the exception OSM.

We found an error in `setLoadFactor(float f)`, a method that was later added to facilitate Korat's test generation [BKM02]. The right part of Figure 6.4 shows the `repOk` OSM of `HashMap`. `repOk` is a predicate used to check class invariants [LG00]. If calling `repOk` on an object state returns `false`, the object state is invalid. By inspecting the `repOK` OSM, we found that calling `setLoadFactor` with any argument value deterministically leads to an invalid state. We checked the source code of `setLoadFactor` and found that its method body is simply `loadFactor = f;`, where `loadFactor` is an object field and `f` is the method argument. The comments for a private field `threshold` states that the value of `threshold` shall be `(int)(capacity`

`* loadFactor)`. Apparently this property is violated when setting `loadFactor` without updating `threshold` accordingly. We fixed this error by appending a call to an existing private method `void rehash()` in the end of `setLoadFactor`'s method body; the `rehash` method updates the `threshold` field using the new value of the `loadFactor` field.

Figure 6.5 shows the `get` OSM of `HashMap`. In the representation of method returns on a transition or in a state, `ret` represents the non-primitive return value and `ret.v` represents the `v` field of the non-primitive return value. Recall that a transition with a dotted edge is exercised only by new generated tests but not by the initial tests. We next walk through the scenario in which developers could inspect Figure 6.5. During inspection, developers might focus their exploration of an OSM on transitions. Three such transitions are `clear`, `remove`, and `put`. Developers are not surprised to see that `clear` or `remove` transitions cause a nonempty `HashMap` to be empty, as is shown by the transitions from the top or bottom state to the central state. But developers are surprised to see the transition of `put(a0:null;a1:null)` from the top state to the central state, indicating that `put` can cause a nonempty `HashMap` to be empty. By browsing the Java API documentation for `HashMap` [SM03], developers can find that `HashMap` allows either a key or a value to be `null`; therefore, the `null` return of `get` does not necessarily indicate that the map contains no mapping for the key. However, in the documentation, the description for the returns of `get` states: "the value to which this map maps the specified key, or null if the map contains no mapping for this key." After reading the documentation more carefully, they can find that the description for `get` (but not the description for the returns of `get`) does specify the accurate behavior. This finding shows that the informal description for the returns of `get` is not accurate or consistent with the description of `get` even in such widely published Java API documentation [SM03].

Figure 6.6 shows a screen snapshot of the `isEmpty` OSM of `HashMap`. We configured Obstra to additionally show each state-preserving transition that has the same method name as another state-modifying transition. We also configured Obstra to display on each edge only the method name associated with the transition. When developers want to see the details of a transition, they can move the mouse cursor over the method name associated with the transition and then the details are displayed. We have searched the Internet for manually created state machines for common data structures but few could be found. One manually created state machine for a container structure [Ngu98] is almost the same as the `isEmpty` OSM of `HashMap` shown in Figure 6.6. There are

two major differences. The $INIT$ state and the `[init]()` transition are shown in Figure 6.6 but not in the manually created state machine. The manually created state machine annotates "not last element" for the state-preserving transition `remove(a0)` (pointed by the mouse cursor in Figure 6.6) on the `isEmpty()=false` state and "last element" for the state-modifying transition `remove(a0)` (shown in the middle of Figure 6.6) starting from the `isEmpty()=false` state; Figure 6.6 shows these two transition names in bold font, indicating them to be nondeterministic. We expect that some of these manually specified conditions for a transition can be inferred by using Daikon [ECGN01] on the variable values collected in the starting state and method arguments for the transition.

6.3.3 Discussion

Our experiences have shown that extracted observer abstractions can help investigate causes of uncaught exceptions, identify weakness of an initial test suite, find bugs in a class implementation or its documentation, and understand class behavior. Although many observer abstractions extracted for the class under test are succinct, some observer abstractions are still complex, containing too much information for inspection. For example, three observers of `HashMap`, such as `Collection values()`, have 43 abstract states. The complexity of an extracted observer abstraction depends on both the characteristics of its observers and the initial tests. To control the complexity, we can display a portion of a complex observer abstraction based on user-specified filtering criteria or extract observer abstractions from the executions of a user-specified subset of the initial tests.

Although the `isEmpty` OSM of `HashMap` is almost the same as a manually created state machine [Ngu98], our approach does not guarantee the completeness of the resulting observer abstractions — our approach does not guarantee that the observer abstractions contain all possible legal states or legal transitions. Our approach also does not guarantee that the observer abstractions contain no illegal transitions. Instead, the observer abstractions faithfully reflect behavior exercised by the executed tests; inspecting observer abstractions could help identify weakness of the executed tests. This characteristic of our approach is shared by other dynamic inference techniques [ECGN01, HD03, WML02, ABL02].

6.4 Conclusion

It is important to provide tool support for developers as they inspect the executions of automatically generated unit tests. We have proposed the observer abstraction approach to aid inspection of test executions. We have developed a tool, called Obstra, to extract observer abstractions from unit-test executions automatically. We have applied the approach on a variety of programs, including complex data structures; our experiences show that extracted observer abstractions provide useful object-state-transition information for developers to inspect.

The preceding chapter discusses a feedback loop between behavior inference and test generation. This chapter shows a type of behavior inference: we infer observer abstractions from the execution of unit tests. The test augmentation in our observer abstraction approach has exploited exercised-concrete-state information inferred from the execution of the initial test suite. Our test generation tools presented in Chapter 4 can be further extended to exploit the inferred observer abstractions to guide their test generation process: given an inferred observer abstraction, the test generation tools can try to generate tests to create new transitions or states in the abstraction. Then the new observer abstraction (inferred from both the initial tests and new tests) can be used to guide the test generation tools to generate tests in the subsequent iteration. Iterations terminate until a user-defined maximum iteration number has been reached or no new transition or state has been inferred from new tests.

Chapter 7

PROGRAM-BEHAVIOR COMPARISON IN REGRESSION TESTING

Regression testing retests a program after it is modified. In particular, regression testing compares the behavior of a new program version in order to the behavior of an old program version to assure that no regression faults are introduced. Traditional regression testing techniques use program outputs to characterize the behavior of programs: when running the same test on two program versions produces different outputs (the old version's output is sometimes stored as the expected output for the test), behavior deviations are exposed. When these behavior deviations are unexpected, developers identify them as regression faults, and may proceed to debug and fix the exposed regression faults. When these behavior deviations are intended, for example, being caused by bug-fixing program changes, developers can be assured so and may update the expected outputs of the tests.

However, an introduced regression fault might not be easily exposed: even if a program-state difference is caused immediately after the execution of a new faulty statement, the fault might not be propagated to the observable outputs because of the information loss or hiding effects. This phenomenon has been investigated by various fault models [Mor90, DO91, Voa92, TRC93]. Recently a *program spectrum* has been proposed to characterize a program's behavior inside the black box of program execution [BL96, RBDL97]. Some other program spectra, such as branch, data dependence, and execution trace spectra, have also been proposed in the literature [BL96, HRS+00, RBDL97].

In this chapter, we propose a new class of program spectra called *value spectra*. The value spectra enrich the existing program spectra family [BL96, HRS+00, RBDL97] by capturing internal program states during a test execution. An internal program state is characterized by the values of the variables in scope. Characterizing behavior using values of variables is not a new idea. For example, Calder et al. [CFE97] propose *value profiling* to track the values of variables during program execution. Our new approach differs from value profiling in two major aspects. Instead of

tracking variable values at the instruction level, our approach tracks internal program states at each user-function entry and exit as the value spectra of a test execution. Instead of using the information for compiler optimization, our approach focuses on regression testing by comparing value spectra from two program versions.

When we compare the dynamic behavior of two program versions, a *deviation* is the difference between the value of a variable in a new program version and the corresponding one in an old version. We compare the value spectra from a program's old version and new version, and use the spectra differences to detect behavioral deviations in the new version[1]. We use a deviation-propagation call tree to show the details of the deviations.

Some deviations caused by program changes might be intended such as by bug-fixing changes and some deviations might be unintended such as by introduced regression faults. To help developers determine if the deviations are intended, it is important to present to developers the correlations between deviations and program changes. A *deviation root* is a program location in the new program version that triggers specific behavioral deviations. A deviation root is among a set of program locations that are changed between program versions. We propose two heuristics to locate deviation roots based on the deviation-propagation call tree. Identifying the deviation roots for deviations can help to understand the reasons for the deviations and determine whether the deviations are regression-fault symptoms or just expected. Identified deviation roots can be additionally used to locate regression faults if there are any.

The next section presents the example that we use to illustrate the definition of value spectra. Section 7.2 presents the value-spectra comparison approach. Section 7.3 describes our experiences of applying the approach on several data structures and then Section 7.4 concludes.

7.1 Example

To illustrate value spectra, we use a sample C program shown in Figure 7.1. This program receives two integers as command-line arguments. The program outputs `-1` if the maximum of two integers is less than `0`, outputs `0` if the maximum of them is equal to `0`, and outputs `1` if the maximum of

[1]Deviation detection in this dissertation is different from the software deviation analysis technique developed by Reese and Leveson [RL97]. Their technique determines whether a software specification can behave well when there are deviations in data inputs from an imperfect environment.

```
#include <stdio.h>
1 int max(int a, int b) {
2     if (a >= b) {
3         return a;
4     } else {
5         return b;
6     }
7 }
8 int main(int argc, char *argv[]) {
9     int i, j;
10    if (argc != 3) {
11       printf("Wrong arguments!");
12       return 1;
13    }
14    i = atoi(argv[1]);
15    j = atoi(argv[2]);
16    if (max(i,j) >= 0){
17        if (max(i, j) == 0){
18            printf("0");
19        } else {
20            printf("1");
21        }
22    } else {
23        printf("-1");
24    }
25    return 0;
26 }
```

Figure 7.1: A sample C program

---

them is greater than `0`. When the program does not receive exactly two command-line arguments, it outputs an error message.

The execution of a program can be considered as a sequence of internal program states. Each internal program state comprises the program's in-scope variables and their values at a particular execution point. Each program execution unit (in the granularity of statement, block, code fragment,

function, or component) receives an internal program state and then produces a new one. The program execution points can be the entry and exit of a user-function execution when the program execution units are those code fragments separated by user-function call sites. Program output statements (usually output of I/O operations) can appear within any of those program execution units. Since it is relatively expensive in practice to capture all internal program states between the executions of program statements, we focus on internal program states in the granularity of user functions, instead of statements.

A *function-entry state* S^{entry} is an internal program state at the entry of a function execution. S^{entry} comprises the function's argument values and global variable values. A *function-exit state* S^{exit} is an internal program state at the exit of a function execution. S^{exit} comprises the function return value, updated argument values, and global variable values. Note that S^{exit} does not consider local variable values. If any of the preceding variables at the function entry or exit is of a pointer type, the S^{entry} or S^{exit} additionally comprises the variable values that are directly or indirectly reachable from the pointer-type variable. A *function execution* $\langle S^{entry}, S^{exit} \rangle$ is a pair of a function call's function-entry state S^{entry} and function-exit state S^{exit}.

Figure 7.2 shows the internal program state transitions of the sample program with the command line arguments of `"0 1"`. In the program execution, the `main` function calls the `max` function twice with the same arguments, and then outputs `"1"` as is shown inside the cloud in Figure 7.2.

7.2 Value-Spectra Comparison Approach

We first introduce a new type of semantic spectra, value spectra, which are used to characterize program behavior (Section 7.2.1). We next describe how we compare the value spectra of the same test on two program versions (Section 7.2.2). We then describe the deviation propagations exhibited by spectra differences (Section 7.2.3). We finally present two heuristics to locate deviation roots based on deviation propagation (Section 7.2.4).

7.2.1 Value Spectra

We propose a new class of semantic spectra, *value spectra*, based on exercised internal program states. Value spectra track the variable values in internal program states, which are exercised as a

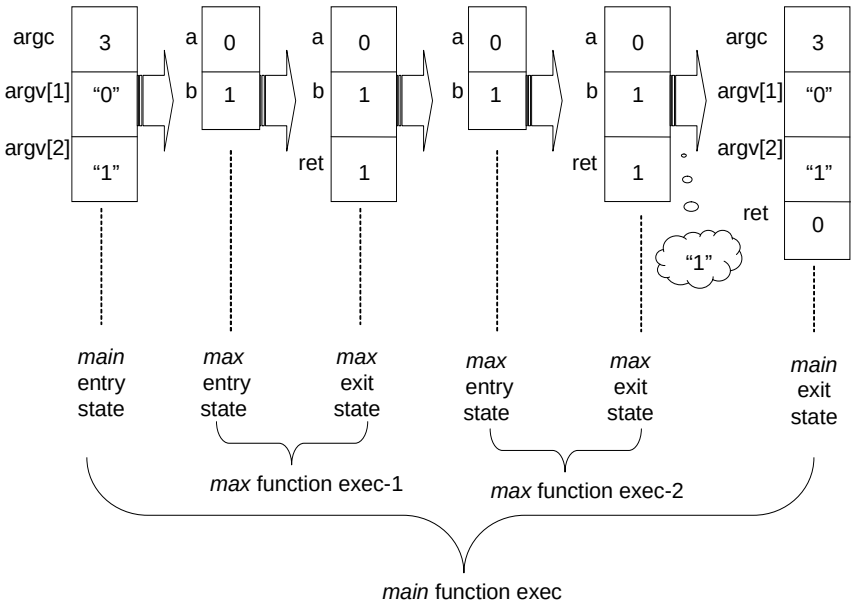

Figure 7.2: Internal program state transitions of the sample C program execution with input `"0 1"`

Table 7.1: Value spectra for the sample program with input `"0 1"`

spectra	profiled entities
value hit	`main(`*`entry`*`(3,"0","1"),`*`exit`*`(3,"0","1",0)),` `max(`*`entry`*`(0,1),`*`exit`*`(0,1,1))`
value count	`main(`*`entry`*`(3,"0","1"),`*`exit`*`(3,"0","1",0))*1,` `max(`*`entry`*`(0,1),`*`exit`*`(0,1,1))*2`

value trace	`main(`*`entry`*`(3,"0","1"),`*`exit`*`(3,"0","1",0)),`
	`max(`*`entry`*`(0,1),`*`exit`*`(0,1,1)), ∨,`
	`max(`*`entry`*`(0,1),`*`exit`*`(0,1,1)), ∨, ∨`

output	`"1"`

program executes.

We propose three new variants of value spectra:

- *User-function value hit spectra* (in short as *value hit spectra*). Value hit spectra indicate whether a user-function execution is exercised.

- *User-function value count spectra* (in short as *value count spectra*). Value count spectra indicate the number of times that a user-function execution is exercised.

- *User-function value trace spectra* (in short as *value trace spectra*). Value trace spectra record the sequence of the user-function executions traversed as a program executes.

Table 7.1 shows different value spectra and output spectra for the sample C program execution with input `"0 1"`. We represent a user-function execution using the following form: `funcname(`*`entry`*`(argvals),`*`exit`*`(argvals,ret))` where `funcname` represents the function name, `argvals` after *`entry`* represents the argument values and global variable values at the function entry, `argvals` after *`exit`* represents the updated argument values and global variable values at the function exit, and `ret` represents the return value of the function. Function executions in value hit spectra or value count spectra do not preserve order, while value trace spectra do preserve order. In value count spectra, a count marker of `"* num"` is appended to the end of each function execution to show that the function execution is exercised `num` times. Note that if we change the second `max` function call from `max(i,j)` to `max(j,i)`, we will have two distinct entities for `max` in the value hit and value count spectra. It is because these two function executions will become distinct with different function-entry or function-exit states. In value trace spectra, `"`$\vee$`"` markers are inserted in the function-execution sequence to indicate function execution returns [RR01]. The value trace spectra for the sample program shows that `main` calls `max` twice. Without these markers, the same function-execution sequence would result from `main` calling `max` and `max` calling `max`.

The value trace spectra strictly subsume the value count spectra, and the value count spectra strictly subsume the value hit spectra. The output spectra are incomparable with any of the three value spectra, since the program's output statements inside a particular user function body might output some constants or variable values that are not captured in that user function's entry or exit state. For example, when we shuffle those `printf` statements in the `main` function body, the program still has the same value spectra but different output spectra. On the other hand, the executions with different value spectra might have the same output spectra. However, when those function bodies containing output statements are not modified in version `P'`, the value trace spectra strictly subsumes the output spectra. In addition, if we also collect the entry and exit states of system output functions in the value trace spectra, the value trace spectra strictly subsume the output spectra.

Value trace spectra strictly subsume dynamically detected invariants because Ernst's Daikon tool [Ern00, ECGN01] generalizes invariants from variable values that define value trace spectra. Because Daikon infers invariants for each function separately and the order among function executions does not affect the inference results, value count spectra also strictly subsume dynamically detected invariants. However, value hit spectra are not comparable to dynamically detected invariants because the number of data samples can affect Daikon's inference results [Ern00, ECGN01]. For example, after we eliminate the second `max` method call by caching the return value of the first `max` method call, we will have the same value count spectra but Daikon might infer fewer invariants for `max` when running the two program versions with input `"0 1"`, because too few data samples exhibit some originally inferred invariants.

Execution-trace spectra strictly subsume any other program spectra, including the three value spectra. Other syntactic spectra, such as branch, path, and data-dependence spectra are incomparable with any of the three value spectra. For example, when we change the statement of `i = atoi(argv[1])` to `i = atoi(argv[1]) + 1`, we will have the same traditional syntactic spectra but different value spectra with input `"0 1"` running on the two program versions. On the other hand, when we move the statement of `printf("1")` from within the inner `else` branch to after the inner `else` branch, and add a redundant statement `i = i + 1` after the `printf("1")` statement, we will have different traditional syntactic spectra, but the same value spectra with input `"0 1"` running on the two program versions.

7.2.2 Value Spectra Differences

To compare program behavior between two program versions, we can compare value spectra from two program versions when we run the same test on them. To compare the value spectra from two program versions, we need to compare function executions from these versions. We can reduce the comparison of two function executions to the comparison of the function-entry and function-exit states from these two function executions, including these states' function names, signatures, and the variable values. When some variables in a function entry or exit state are pointers, their variable values are memory addresses. In the presence of these pointer variables, running a test on the same program twice might produce different value spectra. If we just ignore these pointer-variable

values, we lose the referencing relationships among variables. To address this problem, we perform a linearization algorithm shown in Figure 3.2 of Chapter 3 on each function-entry or function-exit state. In particular, when we encounter a reference-type variable `v`, instead of collecting its value (memory address) in the state representation, we collect the following representation for the variable:

- collect "null" if (`v == null`).
- collect "not_null" if (`v != null`) and there exists no previously encountered variable `v'` such that (`v == v'`).
- collect `vname'` otherwise, where `vname'` is the name of the earliest encountered variable `v'` such that (`v == v'`) and (`v != null`).

Two states S_1 and S_2 are *equivalent* represented as $S_1 \equiv S_2$ if and only if their state representations are the same; otherwise are *nonequivalent*, represented as $S_1 \not\equiv S_2$. Two function executions f_1:$\langle S_1^{entry}, S_1^{exit}\rangle$ and f_2 :$\langle S_2^{entry}, S_2^{exit}\rangle$ are *equivalent* if and only if they have the same function name and signature, $S_1^{entry} \equiv S_2^{entry}$, and $S_1^{exit} \equiv S_2^{exit}$. The comparison of value count spectra additionally considers the number of times that equivalent function executions are exercised. Given a function execution in the new version, the compared function execution in the old version is the one that has the same function name, signature, and function-entry state. If we cannot find such a function execution in the old version, the compared function execution is an *empty function execution*. An empty function execution has a different function name, function signature, function-entry state, or function-exit state from any other regular function executions.

The comparison of value trace spectra further considers the calling context and sequence order in which function executions are exercised. If we want to determine whether two value trace spectra are the same, we can compare the concatenated function-execution sequences of two value traces. If we want to determine the detailed function-execution differences between two value trace spectra, we can use the constructed dynamic call tree and the GNU Diffutils [GNU02] to compare the function-execution traces of two value trace spectra. After the comparison, when a function execution f is present in Version `a` but absent in Version `b`, we can consider that an empty function execution in Version `b` is compared with f in Version `a`.

7.2.3 Deviation Propagation

Assume f_{new}:$\langle S_{new}^{entry}, S_{new}^{exit}\rangle$ is a function execution in a program's new version and f_{old} :$\langle S_{old}^{entry}, S_{old}^{exit}\rangle$ is its compared function execution in the program's old version. If f_{new} and f_{old} are equivalent, then f_{new} is a *non-deviated function execution*. If f_{new} and f_{old} are not equivalent, then f_{new} is a *deviated function execution*. We have categorized a deviated function execution into one of the following two types:

- *Deviation container*. f_{new} is a deviation container, if $S_{new}^{entry} \equiv S_{old}^{entry}$ but $S_{new}^{exit} \not\equiv S_{old}^{exit}$. If a function execution is identified to be a deviation container, developers can know that a certain behavioral deviation occurs *inside* the function execution. Note that when there is a certain behavioral deviation inside a function execution, the function execution might not be observed to be a deviation container, since the behavioral deviation might not be propagated to the function exit.

- *Deviation follower*. f_{new} is a a deviation follower, if $S_{new}^{entry} \not\equiv S_{old}^{entry}$. If a function execution is identified to be a deviation follower, developers can know that a certain behavioral deviation occurs *before* the function execution. For value count spectra particularly, a function execution in a program's new version can be categorized as a deviation follower if its count is different from the count of the compared function execution from the old program version. we need to use a matching technique (similar as the one used in the value trace spectra comparison) to identify which particular function executions in one version are absent in the other version.

The details of value spectra differences can provide insights into deviation propagation in the execution of the new program version. To provide such details, we attach deviation information to a dynamic call tree, where a vertex represents a single function execution and an edge represents calls between function executions. From the trace collected during a test execution, we first construct a dynamic call tree and then annotate the call tree with deviation information to form a deviation-propagation call tree. Figure 7.3 shows the deviation-propagation call trees of two test executions on a new (faulty) version of the `tcas` program. The `tcas` program, its faulty versions, and test suite are contained in a set of `siemens` programs [HFGO94], which are used in the experiment described

```
(The execution of the 58th test)
O main
 |__O initialize
 |__O alt_sep_test
     |__O Non_Crossing_Biased_Climb
     |  |__O Inhibit_Biased_Climb
     |  |__O Own_Above_Threat
     |__O Non_Crossing_Biased_Descend
     |  |__O Inhibit_Biased_Climb
     |  |__O Own_Below_Threat--------[dev follower]
     |  |__O ALIM--------------------[dev follower]
     |__O Own_Above_Threat

(The execution of the 91st test)
O main
 |__O initialize
 |__O alt_sep_test--------------------[dev container]
    |__O Non_Crossing_Biased_Climb
    |  |__O Inhibit_Biased_Climb
    |  |__O Own_Above_Threat
    |  |__O ALIM
    |__O Own_Below_Threat
    |__O Non_Crossing_Biased_Descend-[dev container]
       |__O Inhibit_Biased_Climb
       |__O Own_Below_Threat
```

Figure 7.3: Value-spectra-based deviation-propagation call trees of a new program version (the 9th faulty version) of the `tcas` program

---

in Section 7.3. In the call trees, each node (shown as `O`) is associated with a function execution, and parent node calls its children nodes. For brevity, each node is marked with only the corresponding function name. The execution order among function executions is from the top to the bottom, with the earliest one at the top. If there is any deviated function execution, its deviation type is marked in the end of the function name.

Usually behavioral deviations are originated from certain program locations that are changed in the new program version. These program locations are called *deviation roots*. The function that contains a deviation root is called *deviation-root container*. In the new version of the `tcas` program, a relational operator `>` in the old version is changed to `>=`. The function that contains this changed line is `Non_Crossing_Biased_Descend`.

Some variable values at later points after a deviation-root execution might differ from the ones in the old program version because of the propagation of the deviations at the deviation root. The deviations at the function exit of the deviation-root container might cause the deviation-root container to be observed as a deviation container. Note that some callers of the deviation-root container might also be observed as deviation containers. For example, in the lower call tree of Figure 7.3, the deviation-root container `Non_Crossing_Biased_Descend` is observed as a deviation container and its caller `alt_sep_test` is also observed as a deviation container.

Sometimes deviations after a deviation-root execution might not be propagated to the exit of the deviation-root container, but the deviations might be propagated to the entries of some callees of the deviation-root container, causing these callees to be observed as deviation followers. For example, in the upper call tree of Figure 7.3, the deviation-root container's callees `Own_Below_Threat` and `ALIM` are observed as deviation followers.

7.2.4 Deviation-Root Localization

In the previous section, we have discussed how deviations are propagated given a known deviation root. This section explores the reverse direction: locating deviation roots by observing value spectra differences. This task is called *deviation-root localization*. Deviation-root localization can help developers to better understand which program change(s) caused the observed deviations and then determine whether the deviations are expected.

Recall that given a function execution $f_{new}{:}\langle S_{new}^{entry}, S_{new}^{exit}\rangle$, if f_{new} is a deviation container, S_{new}^{entry} is not deviated but S_{new}^{exit} is deviated; if f_{new} is a deviation follower, S_{new}^{entry} has already been deviated; if f_{new} is a non-deviated function execution, neither S_{new}^{entry} nor S_{new}^{exit} is deviated. Deviation roots are likely to be within those statements executed within a deviation container or before a deviation follower. The following two heuristics are to narrow down the scope for deviation

roots based on deviation propagation effects:

Heuristic 1 Assume `f` is a deviation follower and `g` is the caller of `f`. If (1) `g` is a deviation container or a non-deviated one, and (2) any function execution between `g`'s entry and the call site of `f` is a non-deviated one, deviation roots are likely to be among those statements executed between the `g`'s entry and the call site of `f`, excluding user-function-call statements. For example, in the upper call tree of Figure 7.3, `Own_Below_Threat` is a deviation follower and its caller `Non_Crossing_Biased_Descend` is a non-deviated one. The `Inhibit_Biased_Climb` invoked immediately before the `Own_Below_Threat` is a non-deviated one. Then we can accurately locate the deviation root to be among those statements executed between the entry of `Non_Crossing_Biased_Descend` and the call site of `Own_Below_Threat`.

Heuristic 2 Assume `f` is a deviation container. If any of `f`'s callees is a non-deviated one, deviation roots are likely to be among those statements executed within `f`'s function body, excluding user-function-call statements. For example, in the lower call tree of Figure 7.3, the function execution `Non_Crossing_Biased_Descend` is a deviation container and any of its callees is a non-deviated one. Then we can accurately locate the deviation root to be among those statements executed within the `Non_Crossing_Biased_Descend`'s function body.

When multiple changes are made at different program locations in the new program version, there might be more than one deviation root that cause behavioral deviations. If a deviation root's deviation effect is not propagated to the execution of another deviation root, and each deviation root causes their own value spectra differences, our heuristics can locate both deviation roots at the same time.

7.3 Evaluation

This section presents the experiment that we conducted to evaluate our approach. We first describe the experiment's objective and measures as well as the experiment instrumentation. We then present and discuss the experimental results. We finally discuss analysis cost and threats to validity.

7.3.1 Objective and Measures

The objective of the experiment is to investigate the following questions:

1. How different are the three value spectra types and output spectra type in terms of their deviation-exposing capability?

2. How accurately do the two deviation-root localization heuristics locate the deviation root from value spectra?

Given spectra type S, program P, new version P', and the set CT of tests that cover the changed lines, let $DT(S, P, P', CT)$ be the set of tests each of which exhibits S spectra differences and $LT(S, P, P', CT)$ be the subset of $DT(S, P, P', CT)$ whose exhibited spectra differences can be applied with the two heuristics to accurately locate deviation roots. To answer Questions 1 and 2, we use the following two measures, respectively:

- *Deviation exposure ratio*. The deviation exposure ratio for spectra type S is the number of the tests in $DT(S, P, P', CT)$ divided by the number of the tests in CT, given by the equation: $\frac{|DT(S,P,P',CT)|}{|CT|}$

- *Deviation-root localization ratio*. The deviation-root localization ratio for spectra type S is the number of the tests in $LT(S, P, P', CT)$ divided by the number of the tests in $DT(S, P, P', CT)$, given by the equation: $\frac{|LT(S,P,P',CT)|}{|DT(S,P,P',CT)|}$

Higher values of either measure indicate better results than lower values. In the experiment, we measure the deviation-root localization ratio in the function granularity for the convenience of measurement. That is, when the deviation-root localization locates the deviation-root containers (the functions that contain changed lines), we consider that the localization accurately locates the deviation root. For those changed lines that are in global data definition portion, we consider the deviation-root containers to be those functions that contain the executable code referencing the variables containing the changed data.

7.3.2 Instrumentation

We built a prototype of the spectra-comparison approach to determine the practical utility. Our prototype is based on the Daikon [ECGN01] front end for C programs. Daikon is a system for dynamically detecting likely program invariants. It runs an instrumented program, collects and

examines the values that the program computes, and detects patterns and relationships among those values. The Daikon front end instruments C program code for collecting data traces during program executions. By default, the Daikon front end instruments nested or recursive types (structs that have struct members) with the instrumentation depth of three. For example, given a pointer to the root of a tree structure, we collect the values of only those tree nodes that are within the tree depth of three.

We have developed several Perl scripts to compute and compare all three variants of value spectra and output spectra from the collected traces. In the experiment, we have implemented the deviation-root localization for only value hit spectra.[2] Given two spectra, our tools report in textual form whether these two spectra are different. For value hit spectra, our tools can display spectra differences in deviation-propagation call trees in plain text (as is shown in Figures 7.3) and report deviation-root locations also in textual form.

We use seven C programs as subjects in the experiment. Researchers at Siemens Research created these seven programs with faulty versions and a set of test cases [HFGO94]; these programs are popularly referred as the `siemens` programs (we used the programs, faulty versions, and test cases that were later modified by Rothermel and Harrold [RHOH98]). The researchers constructed the faulty versions by manually seeding faults that were as realistic as possible. Each faulty version differs from the original program by one to five lines of code. The researchers kept only the faults that were detected by at least three and at most 350 test cases in the test suite. Columns 1–4 of Table 7.2 show the program names, number of functions, lines of executable code, and number of tests of these seven subject programs, respectively. Column 5 contains two numbers separated by `"/"`. The first number is the number of the faulty versions selected in this experiment and the second number is the total number of faulty versions. Columns 6 shows the average space cost (in kilobytes) of storing traces collected for a test's value spectra , respectively. The last column shows the description of the subject programs.

We perform the experiment on a Linux machine with a Pentium IV 2.8 GHz processor. In the experiment, we use the original program as the old version and the faulty program as the new version. We use all the test cases in the test suite for each program. To control the scale of the experiment,

[2]We have not implemented deviation-root localization for value count or value trace spectra, because their implementation requires the matching of traces from two versions, which is challenging by itself and beyond the scope of this research.

Table 7.2: Subject programs used in the experiment

program	funcs	loc	tests	vers	|vs_trc|(kb/test)	program description
printtok	18	402	4130	7/7	36	lexical analyzer
printtok2	19	483	4115	10/10	50	lexical analyzer
replace	21	516	5542	12/32	71	pattern replacement
schedule	18	299	2650	9/9	982	priority scheduler
schedule2	16	297	2710	10/10	272	priority scheduler
tcas	9	138	1608	9/41	8	altitude separation
totinfo	7	346	1052	6/23	27	information measure

for those programs with more than 10 faulty versions, we select only those faulty versions in an order from the first version to make each selected version have at least one faulty function that has not yet occurred in previously selected versions.

7.3.3 Results

Figures 7.4 and 7.5 use boxplots to present the experimental results. The box in a boxplot shows the median value as the central line, and the first and third quartiles as the lower and upper edges of the box. The whiskers shown above and below the boxes technically represent the largest and smallest observations that are less than 1.5 box lengths from the end of the box. In practice, these observations are the lowest and highest values that are likely to be observed. Small circles beyond the whiskers are outliers, which are anomalous values in the data.

Figure 7.4 shows the experimental results of deviation exposure ratios that are computed over all subjects. The vertical axis lists deviation exposure ratios and the horizontal axis lists four spectra types: output, value hit, value count, and value trace spectra. Figures 7.5 shows the experimental results of deviation-root localization ratios for value hit spectra. The vertical axis lists deviation-root localization ratios and the horizontal axis lists subject names.

From Figure 7.4, we observed that checking value spectra differences increases the deviation exposure ratio about a factor of three compared to checking program output differences. This indicates that a relatively large portion of deviations could not be propagated to program outputs. There

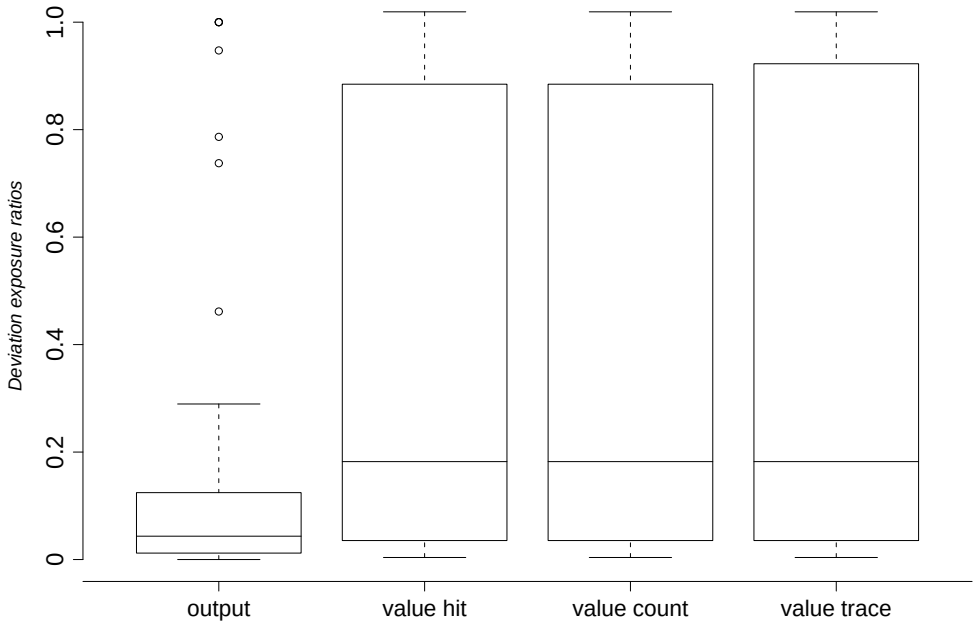

Figure 7.4: Experimental results of deviation exposure ratios

are no significant differences of the deviation exposure ratios among the three value spectra, except that the third quartile of the value trace spectra is slightly higher than the one of the value hit or value count spectra. We found that there were three versions where value trace spectra have higher deviation exposure ratios than value hit and value count spectra. The faults in these three versions sometimes cause some deviation followers to be produced in value trace spectra, but these deviation followers are equivalent to some function executions produced by the old program version; therefore, although the value trace spectra are different, their value hit spectra or value count spectra are the same.

In Figure 7.5, the deviation-root localization ratios for value hit spectra are near 1.0 for all subjects except for the `schedule2` program; therefore, their boxes are collapsed to almost a straight line near the top of the figure. The results show that our heuristics for value hit spectra can accurately locate deviation roots for all subjects except for the `schedule2` program. We inspected `schedule2`'s traces carefully to find out the reasons. We found that the Daikon front end did

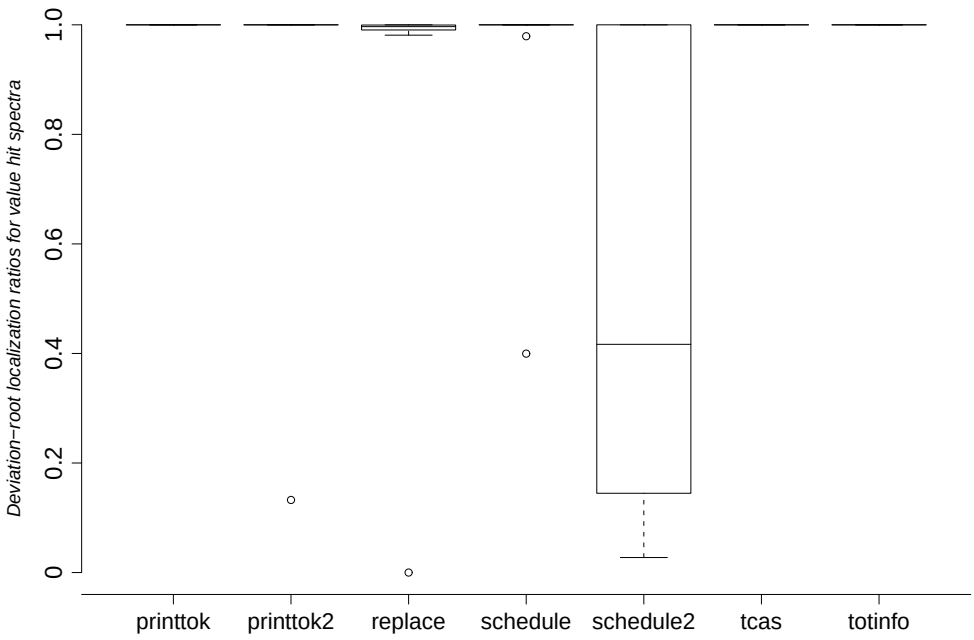

Figure 7.5: Experimental results of deviation-root localization ratios for value hit spectra

not collect complete program state information in a key linked-list struct in `schedule2` using the instrumentation depth of three (the default configuration of the Daikon front end). In some of `schedule2`'s faulty versions, deviations occur on the key linked-list struct beyond the depth of three. Therefore we could not detect the deviations at the exits of deviation roots. We expect that we could increase the deviation-root localization ratios after increasing the instrumentation depth.

The experiment simulates the scenario of introducing regression faults into programs during program modifications. When programmers perform a modification that is not expected to change a program's semantic behavior, such as program refactoring [Fow99], our spectra comparison approach can show the occurrences of unintended deviations and our deviation-root localization accurately locates the regression faults. Moreover, we can reverse the version order by treating the faulty version as the old version and the correct version as the new version. Then we can conduct a similar experiment on them. This simulates the scenario of fixing program bugs. Since our spectra comparison is symmetric, we expect to get the same experimental results. This shows that when

programmers perform a bug-fixing modification, our approach can show them the occurrences of the intended deviations.

7.3.4 Analysis Cost

The space cost of our spectra-comparison approach is primarily the space for storing collected traces. Columns 6 of Table 7.2 shows the average space in kilobytes (KB) required for storing trace of a test's value spectra. The average required space for a test ranges from 8 to 71 KB except for the value spectra of the `schedule` and `schedule2` programs (with the space of 982 and 272 KB, respectively), because these two programs contain global linked-list structs, whose collected values require considerably larger space.

The time cost of our approach is primarily the time of running instrumented code (collecting and storing traces) as well as computing and comparing spectra (deviation-root localization is a part of spectra comparison). The slowdown ratio of instrumentation is the time of running a test on instrumented code divided by the time of running the same test on uninstrumented code. We observed that the average slowdown ratio of instrumentation ranges from 2 to 7, except for the value spectra of `schedule` and `schedule2` programs (with the ratios of 48 and 31, respectively). The average elapsed real time for running a test on instrumented code ranges from 7 to 30 milliseconds (ms), except for the value spectra of `schedule` and `schedule2` programs (with the time of 218 and 137 ms, respectively). The elapsed real time for computing and comparing two spectra of a test ranges from 24 to 170 ms, except for the value spectra of `schedule` and `schedule2` programs (with the time of 3783 and 1366 ms, respectively).

We speculate that applying our approach on larger programs could achieve better improvement of deviation exposure over program output checking, because deviations are probably less likely to be propagated to the outputs of larger programs. We speculate that deviation-root localization ratios based on value spectra might be less affected by the scale of programs than the type of variables used by programs (e.g., simple versus complex data structures).

Larger programs require higher space and time costs. The time or space cost of our value-spectra-comparison approach can be approximately characterized as

$$VCost = O(|vars| \times |userfuncs| \times |testsuite|)$$

where $|vars|$ is the number of variables (including the pointer references reachable from the variables in scope) at the entry and exit of a user function, $|userfuncs|$ is the number of executed and instrumented user functions, and $|testsuite|$ is the size of the test suite.

To address scalability, we can reduce $|testsuite|$ by applying our approach on only those tests selected by regression test selection techniques [RH97]. In addition, we can also reduce $|userfuncs|$ by instrumenting only those modified functions and their (statically determined) up-to-n-level callers or those functions enclosed by identified firewalls [LW90, WL92]. The reduced scope of instrumentation trades a global view of deviation propagation for efficiency.

7.3.5 Threats to Validity

The threats to external validity primarily include the degree to which the subject programs, faults or program changes, and test cases are representative of true practice. The `siemens` programs are small and most of the faulty versions involve simple, one- or two-line manually seeded faults. Moreover, the new versions in our experiment do not incorporate other fault-free changes since all the changes made on faulty versions deliberately introduce regression faults. These threats could be reduced by more experiments on wider types of subjects in future work. The threats to internal validity are instrumentation effects that can bias our results. Faults in our prototype and the Daikon front end might cause such effects. To reduce these threats, we manually inspected the spectra differences on a dozen of traces for each program subject. One threat to construct validity is that our experiment makes use of the data traces collected during executions, assuming that these precisely capture the internal program states for each execution point. However, in practice the Daikon front end explores nested structures up to the depth of only three by default.

7.4 Conclusion

After developers made changes on their program, they can rerun the program's regression tests to assure the changes take effect as intended: refactoring code to improve code quality, enhancing some functionality, fixing a bug in the code, etc. To help developers to gain a higher confidence on their changes, we have proposed a new approach that check program behavior inside the black box over program versions besides checking the black-box program outputs.

We have developed a new class of semantic spectra, called value spectra, to characterize program behavior. We exploit value spectra differences between a program's old version and new version in regression testing. We use these value spectra differences to expose internal behavioral deviations inside the black box. We also investigate deviation propagation and develop two heuristics to locate deviation roots. If there are regression faults, our deviation-root localization additionally addresses the regression fault localization problem. We have conducted an experiment on seven C program subjects. The experimental results show that value-spectra comparison approach can effectively detect behavioral deviations even before deviations are (or even if they are not) propagated to outputs. The results also show that our deviation-root localization based on value spectra can accurately locate the deviation roots for most subjects.

Our approach has not constructed a feedback loop between behavior inference and test generation by using inferred value spectra to guide test generation. However, because generating tests to exhibit program-output deviations in a new version is an undecidable problem, the existing test generation techniques [DO91, KAY98, WE03] for this problem can try to generate tests to propagate the deviation from an outermost deviated function execution to its caller. Through iterations, gradually the value spectra differences can guide the test generation tools to propagate the deviations as close as possible to the program locations for I/O outputs.

Chapter 8

FUTURE WORK

This research has demonstrated that the effectiveness of automated testing can be improved through a framework that reduces the cost of both computer and human effort. There are still many opportunities for extending this work, and this chapter discusses some of the future directions that can be conducted by extending the research in this dissertation.

8.1 Scaling

The experiments that we have conducted in this research primarily focus on unit testing of individual structurally complex data structures. The redundant-test detection approach is evaluated against existing test generation tools, which generate a large number of tests but a relatively small number of non-redundant tests. The non-redundant-test generation and test abstraction approaches are evaluated against a relatively low bound of exhaustive testing. The test selection approach and regression testing approach are evaluated on a set of relatively small programs, being limited in fact by the scalability of the underlying test generation tool or dynamic invariant detection tool (the existing implementation of the regression testing approach uses Daikon's front end to collect value spectra information).

Scaling redundant-test detection deals primarily with reducing the overhead of collecting and storing nonequivalent method executions in memory. For a large program or a test with long method sequences, the size of a single state's representation can be large. For a large test suite, the number of nonequivalent states or method executions can be large. Our implementation employs some state compression techniques such as using a Trie [Fre60] data structure. We can further reduce state-storage requirement by employing some state compression techniques [Hol97, LV01] used in explicit state model checking [CGP99].

Scaling non-redundant-test generation needs to address the same issue in scaling redundant-test

detection: reducing the overhead of keeping track of nonequivalent method executions. We can use those preceding techniques to scale test generation. In addition, we can reduce the state space for exploration in different ways. Instead of exhaustively exploring method sequences (state space) within a small bound, we can explore the state space with longer method sequences with heuristics-guided search [GV02, TAC+04] or evolutionary search [RN95, GK02, Ton04] for achieving certain coverage criteria discussed in Section 2.1. Developers can also specify abstraction functions to reduce the state space (the Rostra techniques based on the `equals` method provide mechanisms for developers to define abstraction functions). Because it may not be feasible to explore a large state space of a single class or multiple classes in a single machine, we can distribute the test generation tasks among multiple machines [MPY+04] and collectively generate tests for a large class or multiple classes in a system. If we use test generation techniques based on concrete or symbolic state exploration, we need to address the communication and coordination issues among multiple machines in avoiding exploring states that have been explored by other machines. If we use test generation techniques based on exploring method sequences without tracking actual concrete or symbolic states, we can get around the communication and coordination issues but with the price of exploring a larger space. In addition, when we test multiple classes in a system (either in a single machine or multiple machines), we need to carefully select the test generation order of multiple classes because we prefer to explore a class `A`'s state space earlier if `A` is an argument type of another class `B`'s method and we want to use the explored states of `A` as the arguments of `B`'s method when exploring `B`'s state space.

Scaling the test selection approach needs to scale both the underlying specification-based test generation tool and dynamic invariant tool. Some techniques for scaling specification-based test generation are similar to those preceding ones for scaling non-redundant-test generation. In addition, we can use some test generation techniques [BKM02] tailored and optimized for exploiting specifications. Some techniques for scaling a dynamic invariant tool has been discussed by Ernst [Ern00] and developed by Perkins and Ernst [PE04]. Scaling our regression testing approach primarily deals with the collection of program state information from test executions and computation of value spectra from program state information. Some techniques for scaling a dynamic invariant tool discussed by Ernst [Ern00] are applicable in addressing the scalability of collecting program state information, such as selectively instrumenting program points.

8.2 New types of behaviors to exploit

Research in this dissertation exploits the inferred behaviors in the form of axiomatic specifications [Hoa69, Gri87] or finite state machines [LY96]. Program behaviors can be described in other forms such as algebraic specifications [GH78] and protocol specifications [BRBY00, BR01, DF01, DLS02], and symmetry properties [Got03]. We can infer these types of behaviors from test executions and use these behaviors to guide test generation by borrowing techniques from specification-based test generation. In addition, we can apply the operational violation approach by selecting any generated tests that violate the behaviors inferred from the existing tests. However, inferring behaviors in the form of algebraic specifications or symmetry properties requires specifically constructed method sequences, which may not already exist in the existing (manually constructed) tests. Therefore, we may need to generate extra new tests to help infer behaviors from the existing tests; the situation is the same in the test abstraction approach: we need to generate extra tests in order to infer observer abstractions from the existing tests.

The operational violation approach selects tests based on a common rationale: selecting a test if the test exercises a certain program behavior that is not exhibited by previously executed tests. We can select tests based on a different new rationale: selecting a test as a special test if the test exercises a certain program behavior that is not exhibited by most other tests; selecting a test as a common test if the test exercises a certain program behavior that is exhibited by all or most other tests. Inferred behaviors in the form of algebraic specifications have been found to be promising for test selection based on this new rationale [Xie04, XN04b].

8.3 New types of quality attributes to test

Our research focuses on testing a program's functional correctness or robustness. We can extend our research to test other quality attributes of a program. For example, software performance testing [AW96, VW98, WV00] creates representative workloads (including average, heavy, or stress workloads) to exercise the program and observe its throughput or response time. In performance testing, generally generating non-redundant tests is still useful to advance program states to reach heavy-loaded states; however, invoking redundant tests sometimes may be useful in performance testing, for example, when a program's performance can be degraded (because of garbage collec-

tion behavior) by running redundant tests that create extensive temporary objects. In performance testing, we can also apply the operational violation approach by inferring a program's performance behavior. Then we can select those generated tests that cause a program to perform worse than the observed performance exercised by the existing tests. We can also infer the characteristics of the bad-performance-inducing tests to help diagnosis the performance-problem roots.

Software security testing [WT03, HM04, PM04] tests a program to make sure the program behave correctly in the presence of a malicious attack. Security risks can be used to guide security testing. For example, for a database application, one potential security risk is SQL injection attacks [HHLT03, HO05]. We can extend our test generation approach to handle complex string operations during symbolic execution. Then we can use symbolic execution to generate test inputs that get through input validators but produce SQL injection attacks. In addition, the operational violation approach has a good potential for security testing, because security testing intends to test the program under malicious inputs, which exercise program behaviors different from the ones exercised by normal inputs in manually created tests.

8.4 Broader types of programs to test

We can detect redundant tests among tests generated for GUI applications [MPS99, Mem01] or directly generate non-redundant tests for GUI applications. In testing GUI applications, event sequences correspond to method sequences in testing object-oriented programs. The program state before or after an event can be abstracted by considering only the state of the associated GUI, which is modeled in terms of the widgets that the GUI contains, their properties, and the values of the properties. Then the techniques of detecting redundant tests or generating non-redundant tests can be similarly applied to GUI tests.

We can detect redundant tests among tests generated for database applications [KS03, CDF+04] and directly generate non-redundant tests for database applications. In testing database applications, the program state before or after a method call additionally includes the database state. After including database states in the program state representation, we can then detect redundant tests for testing database applications. Because a database state can be large, we can use static analysis techniques to determine which parts of the database state are relevant to affect the execution of a method and

consider only these relevant parts when collecting the program state before or after the method call.

We can extend our testing techniques to test programs written in aspect-oriented programming languages such as AspectJ [KLM+97, Tea03]. We can treat an aspect as the unit under test (like a class in an object-oriented program) and advice as the method under test (like a public method in a class). Then we can detect redundant tests for testing an aspect [XZMN04]. In addition, we can adapt our test generation techniques to generate tests for sufficiently exercising an aspect [XZMN05].

Our research focuses on testing a sequential program. When detecting redundant tests for testing a concurrent program, we can no longer operate on the granularity of individual method calls because thread interactions can occur within a method execution causing different method behaviors given the same method inputs. One possible extension to our redundant-test detection techniques is to monitor and collect the inputs to each code segment separated by those thread interaction points within a method. However, this finer granularity can suffer from the state explosion problem more seriously.

8.5 New types of software artifacts to use

This research uses the program under test and sometimes its manually created tests. We can also use other types of software artifacts if they exist in the software development process. For example, if grammars have been written for defining test inputs, we can use these grammars to effectively generate test inputs [SB99, Zay04]. If a method for checking class invariant or a method for validating inputs has been written, we can also use the method to generate test inputs effectively [BKM02]. If requirements are written for the program under test, we can use the requirements to generate tests [WGS94, EFM97, ABM98, GH99]. We can also improve our testing techniques with the information collected from the program's actual usage, such as operational profiling [Woi93, Woi94], or other in-field data [OLHL02, OAH03, MPY+04].

When a model (specification) for a program is specified, model-based testing [DJK+99, GGSV02, Fou, Par04] can be performed. In model-based testing, the underlying model used for test generation is often an abstract one, being derived after abstracting the program's behavior. Two method sequences may produce the same abstract state in the model but we may not want to keep only

one method sequence and discard the other one, because the concrete states (in the code) produced by two method sequences may be different and two method sequences may have different fault-detection capabilities. Although we may not apply redundant-test detection on the generated tests based on abstract states of the model, we can apply redundant-test detection based on the concrete states exercised by tests generated based on the model.

8.6 Testing in the face of program changes

Program changes are inevitable. When a program is changed, rerunning only the tests generated for the old version may not be sufficient to cover the changed or added code, or to expose bugs introduced by the program changes. Although our regression testing techniques intend to exploit the existing tests to expose behavior deviations, generating new tests to exercise the changed or added code is sometimes necessary. Because exploring the whole receiver-object states from the ground for the new version is not economical, we can incorporate incremental computation to re-explore only the parts of the state space that are affected by the program changes.

In general, as has been suggested by longitudinal program analysis [Not02], we can plan and apply test generation across the multitude of program versions. We can use information retained from an earlier test generation to reduce the scope of the test generation on a newer version or to better test a newer version. The way of strategically allocating testing resource might enable us to apply otherwise infeasible test generation over multiple versions of a program as opposed to a specific version.

Chapter 9

ASSESSMENT AND CONCLUSION

This dissertation proposes a framework for improving effectiveness of automated testing in the absence of specifications. A set of techniques and tools have been developed within the framework. First, we have defined redundant tests based on method input values and developed a tool for detecting redundant tests among automatically generated tests; these identified redundant tests increase testing time without increasing the ability to detect faults or increasing developers' confidence on the program under test. Experimental results show that about 90% of the tests generated by the commercial Parasoft Jtest 4.5 [Par03] are redundant tests. Second, we have developed a tool that generates only non-redundant tests by executing method calls symbolically to explore the symbolic-state space. Symbolic execution not only allows us to reduce the state space for exploration but also generates relevant method arguments automatically. Experimental results show that the tool can achieve higher branch coverage faster than the test generation based on concrete-state exploration. Third, we have used Daikon [Ern00] to infer behavior exercised by the existing tests and feed the inferred behavior in the form of specifications to a specification-based test generation tool [Par03]. Developers can inspect those generated tests that violate these inferred behavior, instead of inspecting a large number of all generated tests. Experimental results show that the selected tests have a high probability of exposing anomalous program behavior (either faults or failures) in the program. Fourth, we have used the returns of observer methods to group concrete states into abstract states, from which we construct succint observer abstractions for inspection. An evaluation shows that the abstract-state transition diagrams can help discover anomalous behavior, debug exception-throwing behavior, and understand normal behavior in the class interface. Fifth, we have defined value spectra to characterize program behavior, compared the value spectra from an old version and a new version, and used the spectra differences to detect behavior deviations in the new version. We have further used value spectra differences to locate deviation roots. Experimental results show that comparing value spectra can effectively expose behavior differences between versions even when their

actual outputs are the same, and value spectra differences can be used to locate deviation roots with high accuracy. Finally, putting behavior inference and test generation together, we can construct a feedback loop between these two types of dynamic analysis, starting with an existing test suite (constructed manually or automatically) or some existing program runs. We have shown several instances of the feedback loop in different types of behavior inference. The feedback loop produces better tests and better approximated specifications automatically and incrementally.

9.1 Lessons learned

Software testing research has been conducted for more than three decades. However, when we look at industry, we can find that only a few commercial automated testing tools are available in the market and better tool support is needed in order to meet the demand for high software reliability. The research in this dissertation has developed new techniques and tools to improve the effectiveness of automated software testing. Our work does not assume that the program under test is equipped with specifications, because specifications often do not exist in practice. Our research is motivated to investigate whether benefits of specification-based testing can be achieved to a great extent in the absence specifications and then bring these benefits to a massive group of developers in industry. Our research has shed light on this promising direction and pointed out future work along this direction. In this section, we summarize some lessons that we learned from this research and we hope these lessons may be helpful to other researchers (including us) in pursuing future research.

Dynamic analysis tools can be integrated too. Recently researchers [NE01, Ern03, You03, CS05] have proposed approaches that integrate dynamic and static analysis. Because the results of dynamic analysis based on observed executions may not generalize to future executions, static analysis can be used to verify the results of dynamic analysis [NE01]. Because the results of static analysis may be less precise (more conservative) than what can really occur at runtime, dynamic analysis can be used to select the results of static analysis that can actually occur at runtime [CS05]. Our research shows that dynamic analysis can also be integrated: a dynamic behavior inference tool produces likely properties, which guides a test generation tool to generate tests to violate these properties, and new generated tests are further used to infer new likely properties. A feedback loop on dynamic analysis then can be constructed.

"Both measuring and managing redundancy are important." Our redundant-test detection approach allows us not only to measure test redundancy but also to manage (more precisely, avoid) test redundancy. Although previous research [MK01, Khu03, BKM02, Mar05, VPK04] proposed new techniques for directly generating nonequivalent method inputs (therefore, there is no redundancy among the generated tests), other existing test generation tools may not easily adopt these previously proposed new techniques, partly because these techniques may require specifications or these techniques may not be integrated well with these tools' existing test generation mechanisms. We found it important to measure how well a test-generation tool performs in terms of redundancy among its generated tests, and equally important to guide the tool to improve its performance. Our proposed approach can measure the redundancy of tests generated by *any* test generation tool and compare the performance of different tools based on the measurement results. Indeed, existing test adequacy criteria such as statement coverage can also be used to compare the performance of different tools in terms of satisfying these criteria; however, our proposed measurement offers an operational way of managing (more precisely, avoiding) the test redundancy during or after the tools' existing test generation process.

Breaking into pieces helps. Traditional test-generation techniques consider two tests are different (therefore, both are needed) if these two tests consist of different method sequences; however, it is often expensive to exercise all possible method sequences within even a small sequence-length bound. In fact, we care about the program behavior exercised by each method call individually. After we break a method sequence into pieces of method calls in it, we can check whether at least one of these individual method calls exercise new behavior that has not been exercised before. Breaking the whole into pieces and focusing on pieces instead of the whole can offer opportunities for reducing the space for exploration.

Grouping pieces helps. After the generated tests exercise the concrete state space, the state transition diagram constructed from the whole concrete state is often too complicated to be useful for inspection. After we use an observer method to group together those concrete states whose immediately observable behaviors are the same, we can produce a succinct diagram

for inspection, reducing the human effort in test inspection. In test generation based on state exploration, it is often too expensive to explore the whole concrete state space, our test generation approach then uses symbolic execution to group together those concrete states that can be instantiated from the same symbolic state, reducing the space for exploration.

Looking inside helps. Traditional regression testing techniques look at the observable outputs of two program versions and check whether they are different; however, it is often difficult for existing tests to propagate behavior deviations inside the program executions to the observable outputs. Checking inside the program executions can help expose these behavior deviations even if these deviations are not propagated to the observable outputs. When an object-oriented program is tested, the state of a receiver object can affect the behavior of the method call invoked on the receiver object. As was pointed out by Binder [Bin94], "while limiting scope of effect, encapsulation is an obstacle to controllability and observability of implementation state." Consequently, existing test generation tools consider a receiver object as a black box and invoke different sequences of method calls on the receiver object. However, our research on redundant-test detection and test generation shows that testing tools can still look inside receiver object states at testing time in order to generate tests more effectively.

Exploit the most out of artifacts that already exist in practice. We found that it is a good starting point for tools to first take full advantage of those artifacts that already exist in practice before requiring developers to invest effort in writing extra artifacts solely for the tools. The relatively popular adoption of Parasoft Jtest [Par03] and Agitar Agitator [Agi04] in industry is partly due to their "push button" feature in test automation. At the same time, in order to improve tools' effectiveness, we should exploit the most out of the artifacts that already exist. For example, if an `equals` method exists for a class, our research on redundant-test detection and test generation uses it as an abstraction function to reduce the state space for exploration. Our research on test generation can use the arguments exercised by the manually constructed tests to explore the state space. Our research on test abstraction also uses observer methods of a class as abstraction functions to reduce the state space for inspection. Our research on test selection uses the behavior exercised by the manually constructed tests to guide test

generation and selection.

It is sometimes unavoidable for a tool to ask help from developers (wisely). Our research tries to push up the limit of benefits that automated testing tools can provide; however, we found that we cannot totally leave developers out of the picture, because it is often difficult for the tools to infer the exact intent or expected behavior of the program under test. Our research on test selection, test abstraction, and regression testing produces results for developers to inspect. We found that it is important to allow developers to invest their inspection efforts in an economical way; otherwise, developers would simply give up investing their inspection efforts (thus giving up using the tools). For example, instead of inspecting the output of each single test, developers can inspect a small subset of tests selected by our test selection approach (together with their violated abstractions). Instead of inspecting the complex concrete-state transition diagram, developers can inspect the succinct observer abstractions generated by our test abstraction approach. When presenting information for developers to inspect, tools should be carefully designed to include interesting information as much as possible and at the same time exclude uninteresting information as much as possible.

Working around industrial tools helps. We started the project on test selection for inspection by integrating Daikon [Ern00] and Parasoft Jtest 4.5 [Par03], which is one of a few automated test-generation tool in industry and has a relatively large group of users. Later we started a project on redundant-test detection by detecting a high percentage of redundant tests among tests generated by Parasoft Jtest 4.5. We found that this strategy of working around industrial tools allows a research project to make an impact on industry more easily. Technology transfer or tool adoption in industry is a complex procedure, involving both technical and nontechnical issues. By working around industrial tools, our research can catch industry's attention and facilitate technology transfer by demonstrating that our new techniques can improve existing industrial tools and can be potentially incorporated by them.

Automatically generating complex arguments is more difficult than expected. Test generation techniques based on concrete-state exploration assumes that a set of method arguments are provided and then invokes methods with these arguments to explore the concrete-state space.

Our tool implementation dynamically collects the arguments exercised by a JUnit test class, which is either manually constructed or automatically generated by existing test generation tools. We found that complex arguments generated by existing test generation tools [Par03, CS04] are often not satisfactory when testing some classes that are not data structures. This limitation prevents our test-generation tools from being applied on a significant portion of classes in practice. Although our test generation techniques based on symbolic execution can automatically derive relevant arguments during state exploration, the types of generated arguments are still limited to primitive types. One future solution is to explore the state space of the argument-type objects using method calls. Another solution is to capture and replay the arguments invoked on the class under test when running system tests [SE04, OK05]. Indeed, if class invariants for complex-argument classes exist, some specification-based test-generation tools [MK01, BKM02] can be used to generate valid complex arguments.

Practical lightweight specifications may help. Although our research has developed testing techniques and tools that do not require specifications, we found that the effectiveness of automated testing could be further improved if the tools are given extra guidance in the form of lightweight specifications. In order to make writing specification practical, specifications shall be easy to write and understand. For example, Korat [BKM02, Mar05] generates non-redundant tests by using a `repOk` method, which is an implementation for checking a class invariant [LBR98, LG00]. Tillmann et al. [TS05] proposed an approach that allows developers to write parameterized unit tests, which embed assertions for checking algebraic specifications [GH78]. Then their approach uses symbolic execution to automatically generate relevant arguments for the parameterized unit-test methods.

Model-based testing may be a good way to go when doing integration or system testing. Our research primarily focuses on unit testing. Although some techniques in our research may be adapted to be applied in integration or system testing, integration or system testing in the absence of models (specifications) seems to be more challenging, partly because of the scalability issue. We suspect that developers would be more willing to write models for a whole (sub)system, because the return on investment is much higher than writing specifications for

a class unit. Industrial experiences from Microsoft [GGSV02, Fou] and IBM [Par04] have shown promising results of model-based testing.

Despite the progress we have made in this research, there is much space left for our future work in improving the effectiveness of automated software testing. Our research strategy has been to tackle real but low-end problems where no specifications are assumed, and focus on the units' sequential, functional behaviors (even if structurally complex). When developing techniques and tools for tackling these problems, we learned that the success of automated testing depends on good coordination of effort between computers and developers. Especially when we go beyond low-end problems and try to focus on integration or system testing, non-functional testing, and so on, developers might need to provide significantly more guidance to the tools to improve testing effectiveness.

BIBLIOGRAPHY

[Abb04] Abbot 0.13.1, 2004. `http://abbot.sourceforge.net/`.

[ABL02] Glenn Ammons, Rastislav Bodik, and James R. Larus. Mining specifications. In *Proc. 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, 2002.

[ABM98] Paul E. Ammann, Paul E. Black, and William Majurski. Using model checking to generate tests from specifications. In *Proc. 2rd IEEE International Conference on Formal Engineering Methods*, page 46, 1998.

[ACKM02] G. Alonso, F. Casati, H. Kuno, and V. Machiraju. *Web Services Concepts, Architectures and Applications Series: Data-Centric Systems and Applications*. Addison-Wesley Professional,, 2002.

[AFMS96] David Abramson, Ian Foster, John Michalakes, and Rok Socic. Relative debugging: a new methodology for debugging scientific applications. *Communications of the ACM*, 39(11):69–77, 1996.

[AGH00] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[Agi04] Agitar Agitatior 2.0, Novermber 2004. `http://www.agitar.com/`.

[And79] Dorothy M. Andrews. Using executable assertions for testing and fault tolerance. In *Proc. the 9th International Symposium on Fault-Tolerant Computing*, pages 102–105, 1979.

[AQR+04] Tony Andrews, Shaz Qadeer, Sriram K. Rajamani, Jakob Rehof, and Yichen Xie. Zing: A model checker for concurrent software. In *Proc. 6th International Conference on Computer Aided Verification*, pages 484–487, 2004.

[AW96] Alberto Avritzer and Elaine J. Weyuker. Deriving workloads for performance testing. *Softw. Pract. Exper.*, 26(6):613–633, 1996.

[Bal04] Thomas Ball. A theory of predicate-complete test coverage and generation. Technical Report MSR-TR-2004-28, Microsoft Research, Redmond, WA, April 2004.

[BB04] Clark W. Barrett and Sergey Berezin. CVC Lite: A new implementation of the cooperating validity checker. In *Proc. 16th International Conference on Computer Aided Verification*, pages 515–518, July 2004.

[BCM04] Dirk Beyer, Adam J. Chlipala, and Rupak Majumdar. Generating tests from counterexamples. In *Proc. 26th International Conference on Software Engineering*, pages 326–335, 2004.

[BDLS80] Timothy A. Budd, Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Theoretical and empirical studies on using program mutation to test the functional correctness of programs. In *Proc. 7th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 220–233, 1980.

[Bec00] Kent Beck. *Extreme programming explained*. Addison-Wesley, 2000.

[Bec03] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley, 2003.

[Bei90] Boris Beizer. *Software Testing Techniques*. International Thomson Computer Press, 1990.

[BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Softw. Eng. J.*, 6(6):387–405, 1991.

[BHR+00] Thomas Ball, Daniel Hoffman, Frank Ruskey, Richard Webber, and Lee J. White. State generation and automated class testing. *Software Testing, Verification and Reliability*, 10(3):149–170, 2000.

[Bin94] Robert V. Binder. Object-oriented software testing. *Commun. ACM*, 37(9):28–29, 1994.

[BKM02] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *Proc. International Symposium on Software Testing and Analysis*, pages 123–133, 2002.

[BL96] Thomas Ball and James R. Larus. Efficient path profiling. In *Proc. 29th ACM/IEEE International Symposium on Microarchitecture*, pages 46–57, 1996.

[BL00] Tom Ball and James R. Larus. Using paths to measure, explain, and enhance program behavior. *IEEE Computer*, 33(7):57–65, 2000.

[BMMR01] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proc. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[BOP00] Ugo Buy, Alessandro Orso, and Mauro Pezze. Automated testing of classes. In *Proc. International Symposium on Software Testing and Analysis*, pages 39–48. ACM Press, 2000.

[BPS00] William R. Bush, Jonathan D. Pincus, and David J. Sielaff. A static analyzer for finding dynamic programming errors. *Softw. Pract. Exper.*, 30(7):775–802, 2000.

[BR01] Thomas Ball and Sriram K. Rajamani. Automatically validating temporal safety properties of interfaces. In *Proc. 8th International SPIN Workshop on Model Checking of Software*, pages 103–122, 2001.

[BRBY00] Sergey Butkevich, Marco Renedo, Gerald Baumgartner, and Michal Young. Compiler and tool support for debugging object protocols. In *Proc. 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 50–59, 2000.

[BY01] Luciano Baresi and Michal Young. Test oracles. Technical Report CIS-TR-01-02, University of Oregon, Dept. of Computer and Information Science, Eugene, Oregon, U.S.A., August 2001.

[CDF+04] David Chays, Yuetang Deng, Phyllis G. Frankl, Saikat Dan, Filippos I. Vokolos, and Elaine J. Weyuker. An AGENDA for testing relational database applications. *Softw. Test., Verif. Reliab.*, 14(1):17–44, 2004.

[CDH+00] James C. Corbett, Matthew B. Dwyer, John Hatcliff, Shawn Laubach, Corina S. Pasareanu, Robby, and Hongjun Zheng. Bandera: extracting finite-state models from java source code. In *Proc. 22nd International Conference on Software Engineering*, pages 439–448, 2000.

[CFE97] Brad Calder, Peter Feller, and Alan Eustace. Value profiling. In *Proc. 30th ACM/IEEE International Symposium on Microarchitecture*, pages 259–269, 1997.

[CGP99] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model Checking*. The MIT Press, Cambridge, MA, 1999.

[CGP03] Jamieson M. Cobleigh, Dimitra Giannakopoulou, and Corina S. Pasareanu. Learning assumptions for compositional verification. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 331–346, April 2003.

[Cha82] David Chapman. A program testing assistant. *Commun. ACM*, 25(9):625–634, 1982.

[CL02] Yoonsik Cheon and Gary T. Leavens. A simple and practical approach to unit testing: The JML and JUnit way. In *Proc. 16th European Conference Object-Oriented Programming*, pages 231–255, June 2002.

[Cla76] Lori A. Clarke. A system to generate test data and symbolically execute programs. *IEEE Trans. Softw. Eng.*, 2(3):215–222, 1976.

[Cla00] Mike Clark. Junit primer. Draft manuscript, October 2000.

[CR99] Juei Chang and Debra J. Richardson. Structural specification-based testing: automated support and experimental evaluation. In *Proc. 7th ESEC/FSE*, pages 285–302, 1999.

[CRV94] Yih-Farn Chen, David S. Rosenblum, and Kiem-Phong Vo. Testtube: a system for selective regression testing. In *Proc. 16th International Conference on Software Engineering*, pages 211–220, 1994.

[CS04] Christoph Csallner and Yannis Smaragdakis. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience*, 34:1025–1050, 2004.

[CS05] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *Proc. 27th International Conference on Software Engineering*, pages 422–431, May 2005.

[CTCC98] Huo Yan Chen, T. H. Tse, F. T. Chan, and T. Y. Chen. In black and white: an integrated approach to class-level testing of object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 7(3):250–295, 1998.

[Dai04] Publications using the Daikon invariant detector tool, 2004. `http://www.pag.csail.mit.edu/daikon/pubs-using/`.

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proc. 1st International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, pages 268–284, London, UK, 1993.

[DF94] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3(2):101–130, 1994.

[DF01] Robert DeLine and Manuel Fahndrich. Enforcing high-level protocols in low-level software. In *Proc. ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, pages 59–69, 2001.

[DIS99] Claudio DeMartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. *Softw. Pract. Exper.*, 29(7):577–603, 1999.

[DJK+99] S. R. Dalal, A. Jain, N. Karunanithi, J. M. Leaton, C. M. Lott, G. C. Patton, and B. M. Horowitz. Model-based testing in practice. In *Proc. 21st International Conference on Software Engineering*, pages 285–294, 1999.

[DLP01a] William Dickinson, David Leon, and Andy Podgurski. Finding failures by cluster analysis of execution profiles. In *Proc. 23rd International Conference on Software Engineering*, pages 339–348, 2001.

[DLP01b] William Dickinson, David Leon, and Andy Podgurski. Pursuing failure: the distribution of program failures in a profile space. In *Proc. 8th ESEC/FSE*, pages 246–255, 2001.

[DLS78] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. Hints on test data selection: Help for the practicing programmer. *IEEE Computer*, 11(4):34–41, April 1978.

[DLS02] Manuvir Das, Sorin Lerner, and Mark Seigle. ESP: path-sensitive program verification in polynomial time. In *Proc. ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, pages 57–68, 2002.

[DNS03] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. Technical Report HPL-2003-148, HP Laboratories, Palo Alto, CA, July 2003.

[DO91] Richard A. DeMillo and A. Jefferson Offutt. Constraint-based automatic test data generation. *IEEE Trans. Softw. Eng.*, 17(9):900–910, 1991.

[DvZ03] Markus Dahm and Jason van Zyl. Byte Code Engineering Library, April 2003. `http://jakarta.apache.org/bcel/`.

[ECGN01] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Softw. Eng.*, 27(2):99–123, 2001.

[EFM97] Andre Engels, Loe M. G. Feijs, and Sjouke Mauw. Test generation for intelligent networks using model checking. In *Proc. 3rd International Workshop on Tools and Algorithms for Construction and Analysis of Systems*, pages 384–398, 1997.

[EMR02] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Test case prioritization: A family of empirical studies. *IEEE Trans. Softw. Eng.*, 28(2):159–182, 2002.

[Ern00] Michael D. Ernst. *Dynamically Discovering Likely Program Invariants*. Ph.D., University of Washington Department of Computer Science and Engineering, Seattle, Washington, August 2000.

[Ern03] Michael D. Ernst. Static and dynamic analysis: Synergy and duality. In *Proc. ICSE Workshop on Dynamic Analysis*, pages 24–27, May 2003.

[FL01] Cormac Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. In *Proc. International Symposium of Formal Methods Europe*, pages 500–517, London, UK, 2001.

[Fou] Foundations of Software Engineering at Microsoft Research. The AsmL test generator tool. `http://research.microsoft.com/fse/asml/doc/AsmLTester.html`.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[Fre60] Edward Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, 1960.

[GB03] Erich Gamma and Kent Beck. JUnit, 2003. `http://www.junit.org`.

[GBR98] Arnaud Gotlieb, Bernard Botella, and Michel Rueher. Automatic test data generation using constraint solving techniques. In *Proc. 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 53–62, 1998.

[GG75] John B. Goodenough and Susan L. Gerhart. Toward a theory of test data selection. *IEEE Trans. Software Eng.*, 1(2):156–173, 1975.

[GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *Proc. International Symposium on Software Testing and Analysis*, pages 112–122, 2002.

[GH78] John V. Guttag and James J. Horning. The algebraic specification of abstract data types. *Acta Informatica*, 10:27–52, 1978.

[GH99] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In *Proc. 7th European Software Engineering Conference held jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 146–162, 1999.

[GHK+01] Todd L. Graves, Mary Jean Harrold, Jung-Min Kim, Adam Porter, and Gregg Rothermel. An empirical study of regression test selection techniques. *ACM Trans. Softw. Eng. Methodol.*, 10(2):184–208, 2001.

[GK02] Patrice Godefroid and Sarfraz Khurshid. Exploring very large state spaces using genetic algorithms. In *Proc. 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–280, London, UK, 2002.

[GMH81] John Gannon, Paul McMullin, and Richard Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3(3):211–223, 1981.

[GMS98] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa. Automated test data generation using an iterative relaxation method. In *Proc. 6th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 231–244, 1998.

[GNU02] GNU. GNU diffutils. `http://www.gnu.org/software/diffutils/`, 2002.

[Got03] Arnaud Gotlieb. Exploiting symmetries to test programs. In *Proc. 14th IEEE International Symposium on Software Reliability Engineering*, Denver, Colorado, November 2003.

[GPB02] Dimitra Giannakopoulou, Corina S. Pasareanu, and Howard Barringer. Assumption generation for software component verification. In *Proc. 17th IEEE International Conference on Automated Software Engineering*, pages 3–12, Washington, DC, 2002.

[GPY02] Alex Groce, Doron Peled, and Mihalis Yannakakis. Adaptive model checking. In *Proc. International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 357–370, April 2002.

[Gri87] David Gries. *The Science of Programming*. Springer-Verlag, Secaucus, NJ, USA, 1987.

[GS97] Susanne Graf and Hassen Saidi. Construction of abstract state graphs with PVS. In *Proc. 9th International Conference on Computer Aided Verification*, pages 72–83, 1997.

[Gup03] Neelam Gupta. Generating test data for dynamically discovering likely program invariants. In *Proc. ICSE 2003 Workshop on Dynamic Analysis*, pages 21–24, May 2003.

[GV02] Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. In *Proc. 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 12–21, 2002.

[Ham77] Richard G. Hamlet. Testing programs with the aid of a compiler. *IEEE Trans. Softw. Eng.*, 3(4):279–290, 1977.

[Ham87] Richard G. Hamlet. Probable correctness theory. *Inf. Process. Lett.*, 25(1):17–25, 1987.

[Han03] Hansel 1.0, 2003. `http://hansel.sourceforge.net/`.

[HC01] George T. Heineman and William T. Councill. *Component Based Software Engineering: Putting the Pieces Together*. Addison-Wesley, 2001.

[HD03] Johannes Henkel and Amer Diwan. Discovering algebraic specifications from Java classes. In *Proc. 17th European Conference on Object-Oriented Programming*, pages 431–456, 2003.

[HFGO94] Monica Hutchins, Herb Foster, Tarak Goradia, and Thomas Ostrand. Experiments of the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proc. 16th International Conference on Software Engineering*, pages 191–200, 1994.

[HHLT03] Yao-Wen Huang, Shih-Kun Huang, Tsung-Po Lin, and Chung-Hung Tsai. Web application security assessment by fault injection and behavior monitoring. In *Proc. 12th International Conference on World Wide Web*, pages 148–159, 2003.

[HJL+01] Mary Jean Harrold, James A. Jones, Tongyu Li, Donglin Liang, and Ashish Gujarathi. Regression test selection for java software. In *Proc. 16th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 312–326, 2001.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Gregoire Sutre. Software verification with BLAST. In *Proc. 10th SPIN Workshop on Software Model Checking*, pages 235–239, 2003.

[HL02] Sudheendra Hangal and Monica S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proc. 24th International Conference on Software Engineering*, pages 291–301, 2002.

[HM04] Greg Hoglund and Gary McGraw. *Exploiting Software*. Addison-Wesley, 2004.

[HME03] Michael Harder, Jeff Mellen, and Michael D. Ernst. Improving test suites via operational abstraction. In *Proc. 25th International Conference on Software Engineering*, pages 60–71, 2003.

[HO05] William G.J. Halfond and Alessandro Orso. Combining Static Analysis and Runtime Monitoring to Counter SQL-Injection Attacks. In *Proc. 3rd International ICSE Workshop on Dynamic Analysis*, St. Louis, MO, May 2005. `http://www.csd.uwo.ca/woda2005/proceedings.html`.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hof98] Douglas Hoffman. A taxonomy for test oracles. Tutorial at 11th International Software Quality Week, 1998.

[Hol97] Gerard Holzmann. State compression in Spin. In *Proc. 3rd Spin Workshop*, Twente University, The Netherlands, April 1997.

[Hor02] Susan B. Horwitz. Tool support for improving test coverage. In *Proc. 11th European Symposium on Programming*, pages 162–177, Grenoble, France, April 2002.

[How82] William E. Howden. Weak mutation testing and completeness of test sets. *IEEE Transactions on Software Engineering*, 8(4):371–379, July 1982.

[HRS+00] Mary Jean Harrold, Gregg Rothermel, Kent Sayre, Rui Wu, and Liu Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Journal of Software Testing, Verification and Reliability*, 10(3):171–194, 2000.

[HS96] Merlin Hughes and David Stotts. Daistish: systematic algebraic testing for oo programs in the presence of side-effects. In *Proc. International Symposium on Software Testing and Analysis*, pages 53–61, 1996.

[Hua75] J. C. Huang. An approach to program testing. *ACM Comput. Surv.*, 7(3):113–128, 1975.

[Ios02] Radu Iosif. Symmetry reduction criteria for software model checking. In *Proc. 9th SPIN Workshop on Software Model Checking*, volume 2318 of *LNCS*, pages 22–41. Springer, July 2002.

[JCo03] Jcoverage 1.0.5, 2003. `http://jcoverage.com/`.

[JGS02] Clara Jaramillo, Rajiv Gupta, and Mary Lou Soffa. Debugging and testing optimizers through comparison checking. In Jens Knoop and Wolf Zimmermann, editors, *Electronic Notes in Theoretical Computer Science*, volume 65. Elsevier, 2002.

[Joh74] D. S. Johnson. Approximation algorithms for combinatorial problems. *J. Comput. System Sci.*, 9:256–278, 1974.

[JSS00] Daniel Jackson, Ian Schechter, and Hya Shlyahter. Alcoa: the alloy constraint analyzer. In *Proc. 22nd International Conference on Software Engineering*, pages 730–733, 2000.

[JSS01] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proc. 8th ESEC/FSE*, pages 62–73, 2001.

[KAY96] Bogdan Korel and Ali M. Al-Yami. Assertion-oriented automated test data generation. In *Proc. 18th International Conference on Software Engineering*, pages 71–80. IEEE Computer Society, 1996.

[KAY98] Bogdan Korel and Ali M. Al-Yami. Automated regression test generation. In *Proc. 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 143–152, 1998.

[KBP02] Cem Kaner, James Bach, and Bret Pettichord. *Lessons Learned in Software Testing*. Wiley & Sons, 2002.

[Khu03] Sarfraz Khurshid. *Generating Structurally Complex Tests from Declarative Constraints*. Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, December 2003.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[KJS98] Nathan P. Kropp, Philip J. Koopman Jr., and Daniel P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *Proc. 28th IEEE International Symposium on Fault Tolerant Computing*, pages 230–239, 1998.

[KLM+97] Gregor Kiczales, John Lamping, Anurag Menhdhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. Aspect-oriented programming. In *Proc. 11th European Conference on Object-Oriented Programming*, pages 220–242. 1997.

[Kor90] Bogdan Korel. Automated software test data generation. *IEEE Trans. Softw. Eng.*, 16(8):870–879, 1990.

[KPV03] Sarfraz Khurshid, Corina S. Pasareanu, and Willem Visser. Generalized symbolic execution for model checking and testing. In *Proc. 9th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 553–568, April 2003.

[KS03] Gregory M. Kapfhammer and Mary Lou Soffa. A family of test adequacy criteria for database-driven applications. In *Proc. 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 98–107, 2003.

[KSGH94] David Kung, Nimish Suchak, Jerry Gao, and Pei Hsia. On object state testing. In *Proc. 18th International Computer Software and Applications Conference*, pages 222–227, 1994.

[LBR98] Gary T. Leavens, Albert L. Baker, and Clyde Ruby. Preliminary design of JML: A behavioral interface specification language for Java. Technical Report TR 98-06i, Department of Computer Science, Iowa State University, June 1998.

[LG00] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley, 2000.

[LV01] Flavio Lerda and Willem Visser. Addressing dynamic issues of program model checking. In *Proc. 2001 Spin Workshop*, May 2001.

[LW90] H. K. N. Leung and L. White. A study of integration testing and software regression at the integration level. In *Proc. International Conference on Software Maintenance*, pages 290–300, 1990.

[LY96] David Lee and Mihalis Yannakakis. Principles and methods of testing finite state machines - A survey. In *Proc. The IEEE*, volume 84, pages 1090–1123, August 1996.

[MAD+03] Darko Marinov, Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Martin Rinard. An evaluation of exhaustive testing for data structures. Technical Report MIT-LCS-TR-921, MIT CSAIL, Cambridge, MA, September 2003.

[Mar05] Darko Marinov. *Automatic Testing of Software with Structurally Complex Inputs*. Ph.D., MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, February 2005.

[MBN03] Atif M. Memon, Ishan Banerjee, and Adithya Nagarajan. What test oracle should I use for effective GUI testing? In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 164–173, 2003.

[McM93] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, 1993.

[Mem01] Atif M. Memon. *A Comprehensive Framework for Testing Graphical User Interfaces*. Ph.D., University of Pittsburgh, Pittsburgh, PA, July 2001.

[Mey88] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.

[Mey92] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[MFC01] Tim Mackinnon, Steve Freeman, and Philip Craig. Endo-testing: Unit testing with mock objects. In *Extreme Programming Examined*. Addison-Wesley, 2001.

[MFS90] Barton P. Miller, Louis Fredriksen, and Bryan So. An empirical study of the reliability of unix utilities. *Commun. ACM*, 33(12):32–44, 1990.

[Mic04] Microsoft Visual Studio Developer Center, 2004. `http://msdn.microsoft.com/vstudio/`.

[MK01] Darko Marinov and Sarfraz Khurshid. TestEra: A novel framework for automated testing of Java programs. In *Proc. 16th IEEE International Conference on Automated Software Engineering*, pages 22–31, 2001.

[MKO02] Yu-Seung Ma, Yong-Rae Kwon, and Jeff Offutt. Inter-class mutation operators for Java. In *Proc. International Symposium on Software Reliability Engineering*, pages 352–363, 2002.

[Mor90] L. J. Morell. A theory of fault-based testing. *IEEE Trans. Softw. Eng.*, 16(8):844–857, 1990.

[MPC+02] Madanlal Musuvathi, David Park, Andy Chou, Dawson R. Engler, and David L. Dill. CMC: A pragmatic approach to model checking real code. In *Proc. 5th Symposium on Operating Systems Design and Implementation*, pages 75–88, December 2002.

[MPS99] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Using a goal-driven approach to generate test cases for guis. In *Proc. 21st International Conference on Software Engineering*, pages 257–266, 1999.

[MPS00] Atif M. Memon, Martha E. Pollack, and Mary Lou Soffa. Automated test oracles for guis. In *Proc. 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 30–39, 2000.

[MPY+04] Atif M. Memon, Adam Porter, Cemal Yilmaz, Adithya Nagarajan, Douglas C. Schmidt, and Bala Natarajan. Skoll: Distributed continuous quality assurance. In *Proc. 26th International Conference on Software Engineering*, pages 459–468, 2004.

[MS03] Atif M. Memon and Mary Lou Soffa. Regression testing of guis. In *Proc. 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 118–127, 2003.

[MX05] Amir Michail and Tao Xie. Helping users avoid bugs in GUI applications. In *Proc. 27th International Conference on Software Engineering*, pages 107–116, May 2005.

[Mye79] Glenford J. Myers. *Art of Software Testing*. John Wiley & Sons, Inc., 1979.

[NE01] Jeremy W. Nimmer and Michael D. Ernst. Static verification of dynamically detected program invariants: Integrating Daikon and ESC/Java. In *Proc. 1st Workshop on Runtime Verification*, Paris, France, July 2001.

[NF00] Gleb Naumovich and Phyllis G. Frankl. Toward synergy of finite state verification and testing. In *Proc. 1st International Workshop on Automated Program Analysis, Testing and Verification*, pages 89–94, June 2000.

[Ngu98] Dung Nguyen. Design patterns for data structures. In *Proc. 29th SIGCSE Technical Symposium on Computer Science Education*, pages 336–340, 1998.

[NIS02] National Institute of Standards and Technology. The economic impacts of inadequate infrastructure for software testing. Planning Report 02-3, May 2002.

[Not02] David Notkin. Longitudinal program analysis. In *Proc. 2002 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 1–1, 2002.

[OAFG98] Thomas Ostrand, Aaron Anodide, Herbert Foster, and Tarak Goradia. A visual test development environment for gui systems. In *Proc. 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 82–92, 1998.

[OAH03] Alessandro Orso, Taweesup Apiwattanapong, and Mary Jean Harrold. Leveraging field data for impact analysis and regression testing. In *Proc. 9th European Software Engineering Conference and 10th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 128–137, Helsinki, Finland, September 2003.

[OK05] Alessandro Orso and Bryan Kennedy. Selective Capture and Replay of Program Executions. In *Proc. 3rd International ICSE Workshop on Dynamic Analysis*, St. Louis, MO, May 2005. `http://www.csd.uwo.ca/woda2005/proceedings.html`.

[OLHL02] Alessandro Orso, Donglin Liang, Mary Jean Harrold, and Richard Lipton. Gamma system: Continuous evolution of software after deployment. In *Proc. ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 65–69, Rome, Italy, July 2002.

[Pan78] David J. Panzl. Automatic software test drivers. *Computer*, 11(4):44–50, April 1978.

[Par02] Parasoft. Jcontract manuals version 1.5. Online manual, October 2002. `http://www.parasoft.com/`.

[Par03] Parasoft Jtest manuals version 4.5. Online manual, April 2003. `http://www.parasoft.com/`.

[Par04] Amit M. Paradkar. Plannable test selection criteria for FSMs extracted from operational specifications. In *Proc. 15th International Symposium on Software Reliability Engineering*, pages 173–184, November 2004.

[PE04] Jeff H. Perkins and Michael D. Ernst. Efficient incremental algorithms for dynamic detection of likely invariants. In *Proc. 12th ACM SIGSOFT 12th International Symposium on Foundations of Software Engineering*, pages 23–32, 2004.

[PE05] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *Proc. 19th European Conference on Object-Oriented Programming*, Glasgow, Scotland, July 2005.

[PH90] Karl Pettis and Robert C. Hansen. Profile guided code positioning. In *Proc. ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, pages 16–27, 1990.

[PM04] Bruce Potter and Gary McGraw. Software security testing. *IEEE Security and Privacy*, 2(5):81–85, 2004.

[Pug92] William Pugh. A practical algorithm for exact array dependence analysis. *Commun. ACM*, 35(8):102–114, 1992.

[PVY99] Doron Peled, Moshe Y. Vardi, and Mihalis Yannakakis. Black box checking. In *Proc. International Conference on Formal Description Techniques (FORTE) for Distributed Systems and Communication Protocols, and Protocol Specification, Testing, and Verification (PSTV)*, pages 225–240, October 1999.

[PY99] Christina Pavlopoulou and Michal Young. Residual test coverage monitoring. In *Proc. 21st International Conference on Software Engineering*, pages 277–284, 1999.

[Qui03] Quilt 0.6a, October 2003. `http://quilt.sourceforge.net/`.

[RAO92] Debra J. Richardson, Stephanie Leif Aha, and T. Owen O'Malley. Specification-based test oracles for reactive systems. In *Proc. 14th International Conference on Software Engineering*, pages 105–118, 1992.

[RBDL97] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Proc. the 6th ESEC/FSE*, pages 432–449, 1997.

[RDH03] Robby, Matthew B. Dwyer, and John Hatcliff. Bogor: an extensible and highly-modular software model checking framework. In *Proc. 9th European Software Engineering Conference held jointly with 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 267–276, 2003.

[RDHI03] Robby, Matthew B. Dwyer, John Hatcliff, and Radu Iosif. Space-reduction strategies for model checking dynamic systems. In *Proc. 2003 Workshop on Software Model Checking*, July 2003.

[RH97] Gregg Rothermel and Mary Jean Harrold. A safe, efficient regression test selection technique. *ACM Trans. Softw. Eng. Methodol.*, 6(2):173–210, 1997.

[RHC76] C. Ramamoorthy, S. Ho, and W. Chert. On the automated generation of program test data. *IEEE Trans. Softw. Eng.*, 2(4):293–300, 1976.

[RHOH98] Gregg Rothermel, Mary Jean Harrold, Jeffery Ostrin, and Christie Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *Proc. International Conference on Software Maintenance*, pages 34–43, 1998.

[Ric94] Debra J. Richardson. TAOS: Testing with analysis and oracle support. In *Proc. 1994 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 138–153, 1994.

[RL97] Jon Damon Reese and Nancy G. Leveson. Software deviation analysis. In *Proc. the 19th International Conference on Software Engineering*, pages 250–260, 1997.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice-Hall, Englewood Cliffs, NJ, 1995.

[Rob03] IBM Rational Robot v2003, 2003. `http://www-306.ibm.com/software/awdtools/tester/robot/`.

[Ros92] David S. Rosenblum. Towards a method of programming with assertions. In *Proc. 14th International Conference on Software Engineering*, pages 92–104, 1992.

[RR01] Steven P. Reiss and Manos Renieris. Encoding program executions. In *Proc. 23rd International Conference on Software Engineering*, pages 221–230, 2001.

[RT01] Barbara G. Ryder and Frank Tip. Change impact analysis for object-oriented programs. In *Proc. ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 46–53, 2001.

[RUC01] Gregg Rothermel, Roland J. Untch, and Chengyun Chu. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[RUCH01] Gregg Rothermel, Roland Untch, Chengyun Chu, and Mary Jean Harrold. Prioritizing test cases for regression testing. *IEEE Trans. Softw. Eng.*, 27(10):929–948, 2001.

[SB99] Emin Gun Sirer and Brian N. Bershad. Using production grammars in software testing. In *Proc. 2nd Conference on Domain-Specific Languages*, pages 1–13, 1999.

[SCFP00] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jrapture: A capture/replay tool for observation-based testing. In *Proc. 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 158–167, 2000.

[SE03] David Saff and Michael D. Ernst. Reducing wasted development time via continuous testing. In *Proc. 14th International Symposium on Software Reliability Engineering*, pages 281–292, Denver, CO, November 2003.

[SE04] David Saff and Michael D. Ernst. Automatic mock object creation for test factoring. In *Proc. ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'04)*, pages 49–51, Washington, DC, June 2004.

[SLA02] David Stotts, Mark Lindsey, and Angus Antley. An informal formal method for systematic JUnit test case generation. In *Proc. 2002 XP/Agile Universe*, pages 131–143, 2002.

[SM03] Sun Microsystems. Java 2 Platform, Standard Edition, v 1.4.2, API Specification. Online documentation, Nov. 2003. `http://java.sun.com/j2se/1.4.2/docs/api/`.

[SR02] Hans Schlenker and Georg Ringwelski. POOC: A platform for object-oriented constraint programming. In *Proc. 2002 International Workshop on Constraint Solving and Constraint Logic Programming*, pages 159–170, June 2002.

[ST02] Amitabh Srivastava and Jay Thiagarajan. Effectively prioritizing tests in development environment. In *Proc. International Symposium on Software Testing and Analysis*, pages 97–106, 2002.

[SYC+04] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 133–142, 2004.

[TAC+04] Jianbin Tan, George S. Avrunin, Lori A. Clarke, Shlomo Zilberstein, and Stefan Leue. Heuristic-guided counterexample search in FLAVERS. In *Proc. 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 201–210, 2004.

[Tea03] The AspectJ Team. The AspectJ programming guide. Online manual, 2003.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 119–128, 2004.

[TRC93] Margaret C. Thompson, Debra J. Richardson, and Lori A. Clarke. An information flow model of fault detection. In *Proc. International Symposium on Software Testing and Analysis*, pages 182–192, 1993.

[TS05] Nikolai Tillmann and Wolfram Schulte. Parameterized unit tests. In *Proc. 10th ESEC/FSE*, 2005.

[VHBP00] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In *Proc. 15th IEEE International Conference on Automated Software Engineering*, pages 3–12, 2000.

[Voa92] Jeffrey M. Voas. PIE: A dynamic failure-based technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, 1992.

[VPK04] Willem Visser, Corina S. Pasareanu, and Sarfraz Khurshid. Test input generation with Java PathFinder. In *Proc. 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 97–107, 2004.

[VPP00] William Visser, SeungJoon Park, and John Penix. Using predicate abstraction to reduce object-oriented programs for model checking. In *Proc. 3rd Workshop on Formal Methods in Software Practice*, pages 3–182, 2000.

[VW98] Filippos I. Vokolos and Elaine J. Weyuker. Performance testing of software systems. In *Proc. 1st International Workshop on Software and Performance*, pages 80–87, 1998.

[WE03] Joel Winstead and David Evans. Towards differential program analysis. In *Proc. ICSE 2003 Workshop on Dynamic Analysis*, May 2003.

[Wei99] Mark Allen Weiss. *Data Structures and Algorithm Analysis in Java*. Addison Wesley, 1999.

[WGS94] Elaine Weyuker, Tarak Goradia, and Ashutosh Singh. Automatically generating test data from a boolean specification. *IEEE Trans. Softw. Eng.*, 20(5):353–363, 1994.

[WHLB97] W. Eric Wong, Joseph R. Horgan, Saul London, and Hira Agrawal Bellcore. A study of effective regression testing in practice. In *Proc. 8th International Symposium on Software Reliability Engineering*, pages 230–238, 1997.

[WL92] L. White and H. K. N. Leung. A firewall concept for both control-flow and data-flow in regression integration testing. In *Proc. International Conference on Software Maintenance*, pages 262–271, 1992.

[WML02] John Whaley, Michael C. Martin, and Monica S. Lam. Automatic extraction of object-oriented component interfaces. In *Proc. International Symposium on Software Testing and Analysis*, pages 218–228, 2002.

[Woi93] Denise M. Woit. Specifying operational profiles for modules. In *Proc. International Symposium on Software Testing and Analysis*, pages 2–10, 1993.

[Woi94] Denise M. Woit. *Operational profile specification, test case generation, and reliability estimation for modules*. Ph.D., Queen's University School of Computing, Kingston, Ontario, Canada, February 1994.

[WT03] James A. Whittaker and Herbert H. Thompson. *How to Break Software Security*. Addison-Wesley, 2003.

[WV00] Elaine J. Weyuker and Filippos I. Vokolos. Experience with performance testing of software systems: Issues, an approach, and case study. *IEEE Trans. Softw. Eng.*, 26(12):1147–1156, 2000.

[Xie04] Tao Xie. Automatic identification of common and special object-oriented unit tests. In *Proc. 17th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (Companion)*, pages 324–325, 2004.

[XMN04a] Tao Xie, Darko Marinov, and David Notkin. Improving generation of object-oriented test suites by avoiding redundant tests. Technical Report UW-CSE-04-01-05, University of Washington Department of Computer Science and Engineering, Seattle, WA, Jan. 2004.

[XMN04b] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *Proc. 19th IEEE International Conference on Automated Software Engineering*, pages 196–205, Sept. 2004.

[XMSN05] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Proc. 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 365–381, April 2005.

[XN03a] Tao Xie and David Notkin. Mutually enhancing test generation and specification inference. In *Proc. 3rd International Workshop on Formal Approaches to Testing of Software*, volume 2931 of *LNCS*, pages 60–69, 2003.

[XN03b] Tao Xie and David Notkin. Tool-assisted unit test selection based on operational violations. In *Proc. 18th IEEE International Conference on Automated Software Engineering*, pages 40–48, 2003.

[XN04a] Tao Xie and David Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Proc. 6th International Conference on Formal Engineering Methods*, Nov. 2004.

[XN04b] Tao Xie and David Notkin. Automatically identifying special and common unit tests based on inferred statistical algebraic abstractions. Technical Report UW-CSE-04-08-03, University of Washington Department of Computer Science and Engineering, Seattle, WA, August 2004.

[XN04c] Tao Xie and David Notkin. Checking inside the black box: Regression testing based on value spectra differences. In *Proc. International Conference on Software Maintenance*, pages 28–37, September 2004.

[XZMN04] Tao Xie, Jianjun Zhao, Darko Marinov, and David Notkin. Detecting redundant unit tests for AspectJ programs. Technical Report UW-CSE-04-10-03, University of Washington Department of Computer Science and Engineering, Seattle, WA, October 2004.

[XZMN05] Tao Xie, Jianjun Zhao, Darko Marinov, and David Notkin. Automated test generation for AspectJ program. In *Proc. AOSD 05 Workshop on Testing Aspect-Oriented Programs (WTAOP 05)*, March 2005.

[YE04] Jinlin Yang and David Evans. Dynamically inferring temporal properties. In *Proc. ACM-SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, pages 23–28, 2004.

[You03] Michal Young. Symbiosis of static analysis and program testing. In *Proc. 6th International Conference on Fundamental Approaches to Software Engineering*, pages 1–5, April 2003.

[Zay04] Vadim V. Zaytsev. Combinatorial test set generation: Concepts, implementation, case study. Master's thesis, Universiteit Twente, Enschede, The Netherlands, June 2004.

[ZHM97] Hong Zhu, Patrick A. V. Hall, and John H. R. May. Software unit test coverage and adequacy. *ACM Comput. Surv.*, 29(4):366–427, 1997.

VITA

Tao Xie received his B.S. degree from Fudan University in 1997 and his M.S. degree from Peking University in 2000. In 2000, he started graduate studies at the University of Washington in Seattle, completing a M.S. degree in 2002. After completing his doctoral studies at the University of Washington in 2005, he joined the North Carolina State University (NCSU) as an Assistant Professor in the Department of Computer Science. His primary research interest is software engineering, with an emphasis on techniques and tools for improving software reliability and dependability. He is interested especially in software testing and verification, and broadly in mining of software engineering data, software visualization, program comprehension, software reuse, software metrics, and software evolution.