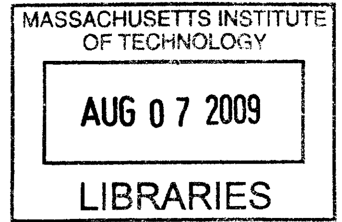
Dynamically Fighting Bugs: Prevention, Detection, and Elimination


MASSACHUSETTS INSTITUTE OF TECHNOLOGY
AUG 07 2009
LIBRARIES

by

Shay Artzi

M.Sc., Technion - Israel Institute of Technology
B.A., Technion - Israel Institute of Technology

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

ARCHIVES

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

© Massachusetts Institute of Technology 2009. All rights reserved.

Author .....
Department of Electrical Engineering and Computer Science
May 21, 2009

Certified by .....
Michael D. Ernst
Associate Professor
Thesis Supervisor

Accepted by .....
Terry P. Orlando
Chairman, Department Committee on Graduate Students

Dynamically Fighting Bugs: Prevention, Detection, and Elimination

by

Shay Artzi

Submitted to the Department of Electrical Engineering and Computer Science
on May 21, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

This dissertation presents three test-generation techniques that are used to improve software quality. Each of our techniques targets bugs that are found by different stake-holders: developers, testers, and maintainers. We implemented and evaluated our techniques on real code. We present the design of each tool and conduct experimental evaluation of the tools with available alternatives.

Developers need to prevent regression errors when they create new functionality. This dissertation presents a technique that helps developers prevent regression errors in object-oriented programs by automatically generating unit-level regression tests. Our technique generates regressions tests by using models created dynamically from example executions. In our evaluation, our technique created effective regression tests, and achieved good coverage even for programs with constrained APIs.

Testers need to detect bugs in programs. This dissertation presents a technique that helps testers detect and localize bugs in web applications. Our technique automatically creates tests that expose failures by combining dynamic test generation with explicit state model checking. In our evaluation, our technique discovered hundreds of faults in real applications.

Maintainers have to reproduce failing executions in order to eliminate bugs found in deployed programs. This dissertation presents a technique that helps maintainers eliminate bugs by generating tests that reproduce failing executions. Our technique automatically generates tests that reproduce the failed executions by monitoring methods and storing optimized states of method arguments. In our evaluation, our technique reproduced failures with low overhead in real programs

Analyses need to avoid unnecessary computations in order to scale. This dissertation presents a technique that helps our other techniques to scale by inferring the mutability classification of arguments. Our technique classifies mutability by combining both static analyses and a novel dynamic mutability analysis. In our evaluation, our technique efficiently and correctly classified most of the arguments for programs with more than hundred thousand lines of code.

Thesis Supervisor: Michael D. Ernst
Title: Associate Professor

Acknowledgments

I greatly enjoyed the five years I spent working on my PhD degree at MIT. I had challenging research problems, excellent advisers, talented collaborators, and great friends. I am grateful to the many people who have contributed to this thesis with their support, collaboration, mentorship, and friendship.

First and foremost thanks go to my adviser, Prof. Michael Ernst. Mike has been an excellent mentor and adviser. His comments were always thoughtful and inspiring. I admire his ability to identify the important problems, and his good judgment in deciding on the most efficient way to address them. I learned much from Mike's guidance and feel very fortunate to have had the opportunity to work with him. I am also grateful to my committee members Prof. Srini Devadas and Dr. Frank Tip for their helpful insights.

During my PhD, I also had the honor to collaborate with IBM researchers. I am grateful to Frank Tip, with whom my interactions have always been a great source of inspiration and helped me develop as a better researcher. Frank always provided me with excellent comments, both for my thesis and for my papers. Special thanks goes to Julian Dolby, who always found the time to help with difficult problems.

During my years in MIT I have been privileged to study and share this work with distinguished professors, students and colleagues. Jeff Perkins was happy to answer my every question, and always provided me with good advice. Sung Kim has an infinite source of great ideas, and an incredible enthusiasm for collaborative research. Collaborating with Sung has been both rewarding and fun. I also thank Yoav Zibin, Noam Shomron, Alexandru Salcianu, Jaime Quinonez, Mapp Papi, Stelios Sidiroglou, Stephen McCamant, Vijay Ganesh, Carlos Pacheco, Angelina Lee, Amy Williams, Derek Rayside, and Danny Dig. Their supportive companionship has been very important to me, and I learned something unique and valuable from each of them.

Adam Kieżun deserves a special thanks. Adam is an amazing researcher and I admire his professionalism, his knowledge, and his ability to solve problems. I learned a lot by observing and collaborating with Adam. Even more important, Adam is a great friend, and was always there when I needed him. Sharing an office with Adam made the entire PhD process easier.

Finally, no acknowledgment is complete without the expression of gratitude towards one's family. I am deeply indebted to my loving wife, Natalie, who has been and continues to be my greatest inspiration. Her confidence in me has kept me going through some very difficult times. I am also very grateful to my parents. Without their support, guidance, and the opportunities they have given me, I could not have achieved this. I want to thank my amazing children, Dolev, Dorin, and Shiri. A true source of happiness even in the hardest times. Finally, I dedicate this dissertation to my dear wonderful wife.

Contents

1 Introduction **11**
1.1 Summary of Contributions 13

2 Prevention **17**
2.1 Technique 19
2.1.1 Model Inference 19
2.1.2 Generating Test Inputs 23
2.2 Evaluation 26
2.2.1 Coverage 26
2.2.2 Palulu Generate Effective Regression Tests 28
2.2.3 Constructing a Complex Input 29
2.3 Related Work 31
2.3.1 Dynamic Call Sequence Graph Inference 31
2.3.2 Generating Test Inputs with a Model 32
2.3.3 Creating Unit Tests from System Tests 32
2.4 Conclusion 33

3 Detection **35**
3.1 Context: PHP Web Applications 39
3.1.1 The PHP Scripting Language 39
3.1.2 PHP Example 39
3.1.3 Failures in PHP Programs 40
3.2 Finding Failures in PHP Web Applications 40
3.2.1 Algorithm 41
3.2.2 Example 41
3.2.3 Path Constraint Minimization 43
3.2.4 Minimization Example 44
3.3 Combined Concrete and Symbolic Execution with Explicit-State Model Checking . 45
3.3.1 Interactive User Simulation Example 45
3.3.2 Algorithm 47
3.3.3 Example 50
3.4 Implementation 51
3.4.1 Executor 52

3.4.2 Bug Finder . . . 54
3.4.3 Input Generator . . . 55
3.5 Evaluation . . . 57
3.5.1 Generation Strategies . . . 57
3.5.2 Methodology . . . 58
3.5.3 Results . . . 59
3.5.4 Threats to Validity . . . 61
3.5.5 Limitations . . . 63
3.6 Related Work . . . 63
3.6.1 Combined Concrete and Symbolic Execution . . . 63
3.6.2 Minimizing Failure-Inducing Inputs . . . 64
3.6.3 Testing of Web Applications . . . 65
3.7 Conclusions . . . 66

4 Elimination **67**
4.1 ReCrash Technique . . . 71
4.1.1 Monitoring Phase . . . 71
4.1.2 Optimizations to the Monitoring Phase . . . 71
4.1.3 Test Generation Phase . . . 74
4.2 Implementation . . . 74
4.2.1 Implementation of the Monitoring Phase . . . 74
4.2.2 Implementation of the Test Generation Phase . . . 78
4.2.3 Optimizations . . . 79
4.3 Experimental Study . . . 80
4.3.1 Subject Systems and Methodology . . . 81
4.3.2 Reproducibility . . . 82
4.3.3 Stored Deep Copy of the Shadow Stack Size . . . 82
4.3.4 Usability Study . . . 82
4.3.5 Performance Overhead . . . 85
4.4 Discussion . . . 86
4.4.1 Limitation in Reproducing Failures . . . 86
4.4.2 Buggy Methods Not in the Stack at Time of the Failure . . . 87
4.4.3 Reusing ReCrash-Generated Tests . . . 88
4.4.4 Privacy . . . 88
4.4.5 Threats to Validity . . . 88
4.5 Related Work . . . 89
4.5.1 Record and Replay . . . 89
4.5.2 Test Generation . . . 90
4.5.3 Remote Data Collection . . . 90
4.6 Conclusion . . . 90

5 Reference Immutability Inference **93**
5.1 Parameter Mutability Definition 95
5.1.1 Informal definition 96
5.1.2 Immutability Classification Example 97
5.1.3 Formal Definition 97
5.1.4 Examples 104
5.2 More Complicated Example of Reference Immutability 105
5.3 Staged Mutability Analysis 107
5.4 Dynamic Mutability Analysis 111
5.4.1 Conceptual Algorithm 111
5.4.2 Optimized Dynamic Analysis Algorithm 112
5.4.3 Dynamic Analysis Heuristics 113
5.4.4 Using Randomly Generated Inputs 114
5.5 Static Mutability Analysis 115
5.5.1 Intraprocedural Points-To Analysis 115
5.5.2 Intraprocedural Phase: S 116
5.5.3 Interprocedural Propagation Phase: P 117
5.6 Evaluation 118
5.6.1 Methodology and Measurements 118
5.6.2 Evaluated Analyses 120
5.6.3 Results 121
5.6.4 Run-time Performance 125
5.6.5 Application: Test Input Generation 126
5.7 Tool Comparison 128
5.7.1 Tools compared 129
5.7.2 Quantitative Comparison 129
5.7.3 Qualitative Comparison 130
5.8 Related Work 136
5.8.1 Discovering Mutability 136
5.8.2 Specifying and Checking Mutability 139
5.9 Conclusion 140

6 Conclusions **143**
6.1 Future Work 144

Chapter 1

Introduction

Software permeates every aspect of our lives. Unfortunately, wherever software is deployed, bugs are sure to follow. A software bug is a defect in a program that prevents it from behaving as intended. A 2002 study of the US Department of Commerce' National Institute of Standards and Technology concluded that software bugs cost the US economy at least 59 billion dollars annually [122]. For instance, software bugs in the Therac-25 radiation therapy machine caused patient deaths [92]. As another example, a prototype of the Ariane 5 rocket was destroyed after launch, due to a bug in the on-board guidance computer program [63].

The software development cycle contains three phases involving code: implementation, testing, and maintenance. Each each of these cycles, different stake-holders need to address issues relating to software bugs. Developers have to prevent new bugs, testers try to detect hidden bugs, and maintainers need to eliminate manifested bugs. In this dissertation we present techniques to prevent, detect, and eliminate software bugs during these three phases. All our techniques achieve better results than previous techniques for the same problems. Each of our techniques automatically generates tests to address software bugs based on how and by whom the bugs get discovered:

- During the implementation phase, it is the developer's task to prevent new software bugs from being introduced into the software. We present a technique that automatically generates tests to prevent software regressions.

- During the testing phase, it is the tester's task to detect hidden software bugs. We present a technique that automatically generates tests exposing failures in the software.

- During the maintenance phase, it is the maintainer's task to eliminate software bugs discovered by users. We present a technique that automatically generates tests that reproduce failures to help eliminate software bugs.

In addition, this thesis also introduces a combined static and dynamic[1] analysis for classifying the mutability of references. The mutability classification increases the effectiveness of our prevention

[1]Dynamic analysis observes the behavior of the program when it is executed to provide information such as execution traces, coverage information, and timing profiles. Static analysis examines the program's source code without executing it, for various kinds of information

technique and the performance of our elimination technique.

There are previous several solutions for handling software bugs. Each of our technique builds on, and improves such previous solutions.

Prevention A regression error occurs when working functionality ceases to work as intended, after a program change. A common way to prevent regression errors is by using regression tests. Regression tests exercise existing functionally while validating the expected results. When a regression test fails, the developer can either fix the regression error or update the test. Unit level regression tests any very useful. The execution time of unit level regression tests is fast, and they make the tasks of locating regression errors easy. System level regression tests, on the other hand, can take hours or days to execute. Thus, regression errors are discovered a long time after being introduced, and locating them becomes much harder. Unfortunately, since creating unit level regression tests is time consuming, they only exercise small parts of the functionality for most programs, while system level regression tests exercise more functionality.

We present a technique that automatically generates regression tests on the unit level. The technique creates models of program behavior from example executions of the program. Then the technique generates unit level regression tests for the program by using the models to guide random test generation.

Detection Traditional testing is aimed at detecting bugs by executing the software under controlled conditions and evaluating the results. The goal of the testing is to increase the confidence of the tester that the software behaves as expected. However, due to the nature of the programming language or the output, the tests are often lacking in either code coverage, or specifications coverage.

We present a technique that generate tests to automatically detect failures in the domain of dynamic web applications. In this domain tests are harder to create manually due to the dynamicity of underlying languages, the frequency with which the code is modified, and the heavy dependence of the program flow on user interaction. Our technique combines dynamic test generation [27, 67, 133] with explicit state model-checking, which were not exploited before for the task of locating bugs in dynamic web applications.

Elimination Debugging is the process of finding and eliminating bugs. First, the developer needs to be able to reproduce the failure consistently. Once the failure can be reproduced, the developer can find the causing bug, and then try a solution for the bug. The task of reproducing a failure becomes harder when the failure is discovered by a user in a deployed application during the maintenance phase.

We present an approach to reproduce failures efficiently, without introducing any changes to the user's environment, and with low execution overhead. This approach connects debugging with testing by storing partial information on method entry, and generating units test that reproduce an observed failure.

In this dissertation we have found that dynamic analysis is a very useful tool in addressing problems related to software bugs, especially when combined with other types of analysis. Our

prevention technique uses dynamic analysis to create models of program execution and combines dynamic analysis with randomized test generation. This combination has not been previously used for creating regression tests. Our detection technique uses dynamic analysis (as part of the dynamic test generation) to find the correlation between inputs and the paths taken during the execution. In order to explore more functionally, our technique combines the dynamic analysis with explicit state model checking to allow the simulation of multi-stage user interaction. This allows the dynamic test generation to be effective for the domain of web-applications in which we are the first to have tried it. Our elimination technique uses dynamic analysis to store information during the execution. This information is used to revert the system to different states from before the failure happened. We combine the dynamic analysis with static analysis used to reduce the overhead of storing the information. Finally, our mutability inference technique combines scalable dynamic analysis inference with the accurate lightweight static analysis inference to achieve a combined scalable and accurate technique.

The different uses of the generated tests also determine the similarity of the tests to the observed executions. When our technique generates tests to prevent bugs in existing working software, the goal of the tests is to exercise functionality which is similar to the observed existing functionality. When our technique generates tests to expose hidden bugs in the program, the goal of the tests is to explore unrestricted different functionality. Finally, when our technique generates tests for reproducing a failure, the goal of the tests is the imitate as closely as possible the original failing execution.

Section 1.1 presents a detailed list of the contributions in this thesis.

1.1 Summary of Contributions

This dissertation presents the following contributions:

- A technique that combines dynamic analysis with random testing to automatically generate regression tests in order to reduce the number of new bugs introduced into object oriented programs when the program is modified. An earlier version of this technique appeared in [10].

  The technique uses an example execution of the program to infer a model of legal call sequences. The model guides a random input generator towards legal but behaviorally-diverse sequences that serve as legal test inputs. This technique is applicable to programs without formal specification, even for programs in which most sequences are illegal.

  We present an implementation of this technique, in a tool called Palulu, and evaluate its effectiveness in creating legal inputs for real programs. Our experimental results indicate that the technique is effective and scalable. Our preliminary evaluation indicates that the technique can quickly generate legal sequences for complex inputs: in a case study, Palulu created legal test inputs in seconds for a set of complex classes, for which it took an expert thirty minutes to generate a single legal input.

- A technique that leverages the combined concrete and symbolic execution technique, to automatically expose and localize crashes and malformed dynamically-generated web pages in dynamic web applications. These common problems seriously impact usability of Web applications. Current tools for Web-page validation cannot handle the dynamically-generated pages that are ubiquitous on today's Internet. An earlier version of this technique appeared in [11].

  We present a dynamic test generation technique for the domain of dynamic web applications. The technique utilizes both combined concrete and symbolic execution and explicit-state model checking. The technique generates tests automatically, runs the tests capturing logical constraints on inputs, and creates failing tests. It also minimizes the conditions on the inputs to failing tests, so that the resulting bug reports are small and useful in finding and fixing the underlying faults.

  Our tool Apollo implements the technique for the PHP programming language. Apollo generates test inputs for a web application, monitors the application for crashes, and validates that the output conforms to the HTML specification. We present Apollo's algorithms and implementation, and an experimental evaluation that revealed 302 faults in 6 PHP web applications.

- A technique that uses dynamic analysis to reproduce software failures in object-oriented programs. The technique automatically converts a crashing program execution into a set of deterministic and self-contained unit tests. Each of the unit tests reproduces the problem from the original program execution. An earlier version of this technique appeared in [13].

  The technique is based on the idea that it is not necessary to replay the entire execution in order to reproduce a specific crash. For many crashes, creating a test case requires only partial information about the methods currently on the stack. Our technique exploits the premise of unit testing (that many bugs are dependent on small parts of the heap) and the fact that good object-oriented style avoids excessive use of globals. The technique is efficient, incurring low performance overhead. The technique has a mode, termed "second chance" in which the technique will have negligible overhead until a crash occurs and very small overhead until the crash occurs for a second time, at which point the test cases are generated.

  We present ReCrashJ, an implementation of our approach for Java. ReCrashJ reproduced real crashes from Javac, SVNKit, Eclipsec, and BST. ReCrashJ is efficient, incurring 13%–64% performance overhead. In "second chance" mode, ReCrashJ had a negligible overhead until a crash occurs and 0%–1.7% overhead until the crash occurs for a second time, at which point the test cases are generated.

- A technique that uses a combination of static and dynamic analysis to detect method parameters that will not change during execution. An earlier version of this technique appeared in [14].

  Knowing which method parameters may be mutated during a method's execution is useful for many software engineering tasks. A parameter reference is *immutable* if it cannot be used to modify the state of its referent object during the method's execution. We formally

define this notion, in a core object-oriented language. Having the formal definition enables determining correctness and accuracy of tools approximating this definition and unbiased comparison of analyses and tools that approximate similar definitions.

We present Pidasa, a tool for classifying parameter reference immutability. Pidasa combines several lightweight, scalable analyses in stages, with each stage refining the overall result. The resulting analysis is scalable and combines the strengths of its component analyses. As one of the component analyses, we present a novel dynamic mutability analysis and show how its results can be improved by random input generation. Experimental results on programs of up to 185 kLOC show that, compared to previous approaches, Pidasa improves both the run-time performance and the overall accuracy of immutability inference. Mutability classifications are used in our first technique (prevention) to reduce the size of the generated models, and in our third technique (elimination) to reduce the amount of information stored on method entry.

The rest of the dissertation is organized as follows. Chapters 2–5 discuss the main parts of the thesis, introduce the problems and algorithms by examples, and present the design of the experiments that evaluate the work. The chapters correspond to the main components of the work—bug prevention by automatically creating regression tests (Chapter 2), bug detection in dynamic web applications (Chapter 3), bug elimination using stored object states (Chapter 4), and mutability analysis infrastructure (Chapter 5). Each of the chapters is self contained and can be read in any order. Finally, Chapter 6 summaries this dissertation and presents several directions for future work.

Chapter 2

Prevention

This chapter presents a test generation technique to help developers prevent software regressions. A software regression occurs when existing working functionality ceases to work as a result of software changes.

Regression testing is commonly used to prevent software regressions. Regression testing involves the creation of a set of test cases, covering as much of the software functionality as possible, and the frequent execution of the tests during development. Working on software that has minimal or no regression tests is like walking on eggshells: every change can break something. However, unlike walking on eggshells, the damage is not always immediately obvious.

The task of creating regression tests is time-consuming. As a result of limited resources, developers often neglect the creation of regression tests, or more typically create a set of tests that doesn't cover most of the existing functionality.

In this chapter we present an automated way of creating test inputs for object- oriented programs [40, 110, 149, 156, 157]. A test input for an object-oriented program typically consists of a sequence of method calls that use the public interface (API) defined by the program under test. For example, `List l = new List(); l.add(1); l.add(2)` is a test input for a class that implements a list. Tests inputs can be automatically or manually augmented with assertions to create regression tests. For many programs, most method sequences are illegal: for correct operation, calls must occur in a certain order with specific arguments. Techniques that generate unconstrained sequences of method calls without the use of formal specifications are bound to generate mostly illegal inputs.

For example, Figure 2-1 shows a test input for the tinySQL database server[1]. Before a query can be issued, a driver, a connection, and a statement must be created, and the connection must be initialized with a meaningful string (e.g., "`jdbc : tinySQL`"). As another example, Figure 2-8 shows a test input for a more complex API.

Model-based testing [29, 35, 52, 70, 73, 76, 108, 121, 144, 147] offers one solution. A model can specify legal method sequences (e.g., `close()` cannot be called before `open()`, or `connect()` must be called with a string that starts with "`jdbc :`"). But as with formal specifications, most programmers are not likely to write models (except perhaps for critical components), and thus non-critical code may not take advantage of model-based input generation techniques.

[1] http://sourceforge.net/projects/tinysql

```
TextFileDriver d = new TextFileDriver();
Conn con = d.connect("jdbc:tinySQL",null);
Stmt s2 = con.createStmt();
s2.execute("CREATE TABLE test (name char(25), id int)");
s2.executeUpdate("INSERT INTO test(name, id) VALUES('Bob', 1)");
s2.close();
Stmt s1 = con.createStmt();
s1.execute("DROP TABLE test");
s1.close();
con.close();
```

Figure 2-1: Example of a manually written client code using the tinySQL database engine. The client creates a driver, connection, and statements, all of which it uses to query the database.

To overcome the problem of illegal inputs, we developed a technique that combines dynamic model creation and testing. Our technique creates a model of method sequences from an example execution of the program under test, and uses the model to guide a test input generator towards the creation of legal method sequences. Because the model's sole purpose is aiding an input generator, our model inference technique is different from previous techniques [7, 36, 154, 161] which are designed primarily to create small models for program understanding. Our models must contain information useful for input generation, and must handle complexities inherent in realistic programs (for example, nested method calls) that have not been previously considered. At the same time, our models need not contain any information that is useless in the context of input generation such as methods that do not mutate state.

Our generator uses the model to *guide* its input generation strategy. The emphasis on "guide" is key: to create behaviorally diverse inputs, the input generator may diverge from the model, which means that the generated sequences are similar to, but not identical to, the sequences used to infer the model. Generating such sequences is desirable because it permits our test generation technique to construct new behaviors rather than merely repeating the observed ones. Our technique creates diverse inputs by (i) generalizing observed sequences (inferred models may contain paths not observed during execution), (ii) omitting certain details from models (e.g., values of non-primitive, non-string parameters), and (iii) diverging from models by randomly inserting calls to methods not observed during execution. (iv) combining instances created from different sequences to further explore possible functionality. Some of the generated inputs may be illegal—our technique uses heuristics that discard inputs that appear to be illegal based on the result of their execution [111].

In this chapter, we present the following contributions:

- We present a dynamic model-inference technique that infers call sequence models suitable for test input generation. The technique handles complexities present in real programs such as nested method calls, multiple input parameters, access modifiers, and values of primitives and strings.
- We present a test-input generation technique that uses the inferred models, as well as feedback obtained from executing the sequences, to guide generation towards legal, non-trivial

sequences.

- We present Palulu, a tool that implements both techniques for Java. The input to palulu is a program under test and an example execution. Palulu uses the example execution to infer a model, then uses the model to guide the input generation in creating inputs. Palulu's output is a collection of test inputs for the program under test.
- We evaluate Palulu on a set of real applications with constrained interfaces, showing that the inferred models assist in generating inputs for these programs. We also show that tests created using our models on highly constrained object oriented programs achieve better coverage than tests created using the universal models.

The remainder of the chapter is organized as follows. Section 2.1 presents the technique. Section 2.2 describes an experimental evaluation of the technique. Section 2.3 surveys related work, and Section 2.4 concludes the chapter

2.1 Technique

The input to our technique is an example execution of the program under test. The output is a set of test inputs for the program under test. The technique has two steps. First, it infers a model that summarizes the sequences of method calls (and their input arguments) observed during the example execution. Section 2.1.1 describes model inference. Second, the technique uses the inferred models to guide input generation. Section 2.1.2 describes test input generation.

2.1.1 Model Inference

For each class observed during execution, our technique constructs a model called a *call sequence graph*. Call sequence graphs are rooted, directed, and acyclic. The edges represent method calls and their primitive and string arguments. Each node in the graph represents a collection of object states, each of which may be obtained by executing the method calls along some path from the root to the node. In other words, a node describes the history of calls. Each path starting at the root corresponds to a sequence of calls that operate on a specific object—the first method constructs the object, while the rest of the methods mutate the object (possibly as one of their parameters). Note that when two edges point to the same node, it does not necessarily mean that the underlying state of the program is the same.

The call sequence graph (model) for the class is a summary of call sequence graphs for all instances of the class. The model inference algorithm constructs a model in two steps. First, it constructs a call sequence graph for each instance of the class, observed during execution. Second, it creates the model for the class by merging all the call sequence graphs of instances of the class.

2.1.1.1 Constructing the Call Sequence Graph

A *call sequence* of an object contains all the calls in which the object participated as the receiver or a parameter, with the method nesting information for sub-calls. Figure 2-2(b) shows a call

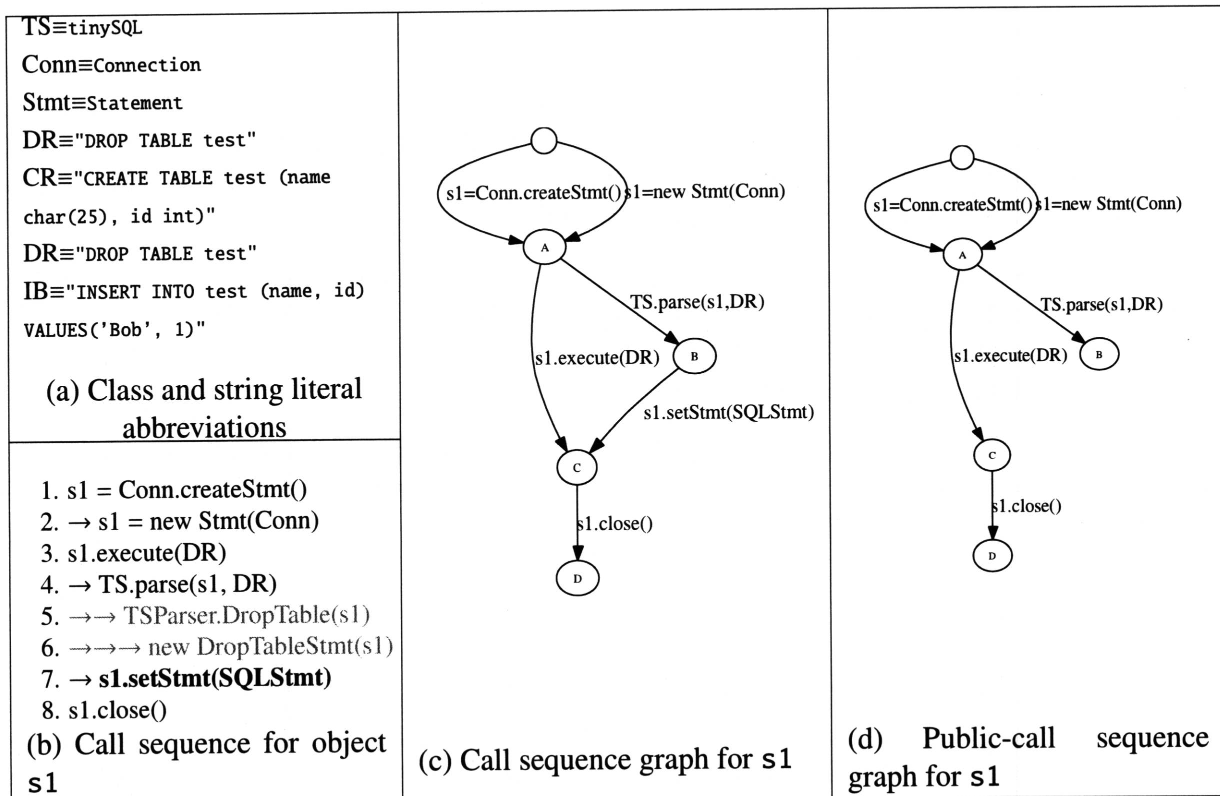


Figure 2-2: Constructing a call sequence graph for an object. (a) Abbreviations used in Figures 2-2 and 2-4. (b) Call sequence involving object `s1` in the code from Figure 2-1. Indented lines (marked with arrows) represent nested calls, grey lines represents state-preserving calls, and lines in bold face represent non-public calls. (c) Call sequence graph for `s1` inferred by the model inference phase; it omits state-preserving calls. The path A-B-C represents two calls (lines 4 and 7) nested in the call in line 3. (d) Public call sequence graph, after removing from (b) an edge corresponding to a non-public call.

sequence of the `Stmt` instance `s1` from Figure 2-1. A *call sequence graph* of an instance is a graph representation of the object's call sequence—each call in the sequence has a corresponding edge between some nodes, and calls nested in the call correspond to additional paths between the same nodes. Edges are annotated with primitive and string arguments of the calls, collected during tracing. (Palulu records method calls, including arguments and return values, and field/array writes in a trace file created during the example execution of the program under test.)

The algorithm for constructing an object's call sequence graph has three steps. First, the algorithm removes state-preserving calls from the call sequence. Second, the algorithm creates a call sequence graph from the call sequence. For nested calls, the algorithm creates alternative paths in the graph. Third, the algorithm removes non-public calls from the graph.

Figure 2-2 (c) shows the call sequence graph corresponding to the call sequence in Figure 2-2 (b). The call sequence graph indicates, for example, that it is possible to transition an instance of `Stmt` from state A to state C either by calling `s1.execute()` or by calling `TS.parse(s1, DR)` and then

calling `s1.setStmt(SQLStmt)`. Figure 2-2 (d) shows the final call sequence graph for the instance `s1`, after removing non-public calls.

Next, we describe the steps in the algorithm for constructing an object's call sequence graph:

1. Removing state-preserving calls. The algorithm removes from the call sequence all calls that do not modify the state of the object.

State-preserving calls are of no use in constructing inputs, and omitting them reduces model size and search space without excluding any object states. Use of a smaller model containing only state-changing calls makes test generation more likely to explore many object states (which is one goal of test generation) and aids in exposing errors. State-preserving calls can, however, be useful as oracles for generated inputs, which is another motivation for identifying them. For example, the call sequence graph construction algorithm ignores the calls in lines 5 and 6 in Figure 2-2(b).

To discover state-preserving calls, the technique use the immutability analysis technique (Chapter 5) on the subject program. A method parameter (including the receiver) is considered immutable if no execution of the method changes the state of the object passed to the method as the actual parameter. The "state of the object" is the part of the heap that is reachable from the object by following field references.

2. Constructing call sequence graph. The call sequence graph construction algorithm is recursive and is parameterized by the call sequence, a starting node, and an ending node. The top-level invocation (for the entire history of an object) uses the root as the starting node and a dummy node as the ending node[2].

Figure 2-3 shows a pseudo-code implementation of the algorithm. The algorithm processes the call sequence call by call, while keeping track of the last node it reached. When a call is processed, a new edge and node are created and the newly created node becomes the last node. The algorithm annotates the new edge with the primitive and string arguments of the call.

Nested calls are handled by recursive invocations of the construction algorithm and give rise to alternate paths in the call sequence graph. After a call to method c is processed (i.e., an edge between nodes n_1 and n_2 is added to the graph), the algorithm creates a path in the graph starting from n_1 and ending in n_2, containing all calls invoked by c.

For example, Figure 2-2(c) contains two paths from state A to state C. This alternative path containing `TS.parse(s1, DR)` and `s1.setStmt(SQLStmt)` was added because the call to `s1.execute()` (line 3) of Figure 2-2(b) invokes those two calls (lines 4 and 7).

3. Removing non-public calls. After constructing the object's call sequence graph, the algorithm removes from the graph each edge that corresponds to a non-public method. Thus, each path through the graph represents a sequence of method calls that a client (such as a test case) could make on the class. Non-public calls are removed from the graph after it is constructed because removing them in the same way as state-preserving calls would create paths that were never seen in the original execution.

For example, Figure 2-2(d) presents the call sequence graph after removing the edge corresponding to the non-public method `s1.setStmt(SQLStmt)` in Figure 2-2(c).

[2]Dummy nodes are not shown in Figures 2-2 and 2-4.

```
// Insert sequence cs between nodes start and end.
createCallSequenceGraph(CallSequence cs, Node start, Node end) {
  Node last = start;
  for (Call c : cs.topLevelCalls()) {
    Node next = addNewNode();
    addEdge(c, last, next); // add "last --c--> next"
    CallSequence nestedCalls = cs.getNestedCalls(c);
    createCallSequenceGraph(nestedCalls, last, next);
    last = next;
  }
  replaceNode(last, end); // replace last by end
}
```

Figure 2-3: The call sequence graph construction algorithm written in Java-like pseudo-code. The algorithm is recursive, creating alternate paths in the graph for nested calls. The algorithm uses the following auxiliary functions. `topLevelCalls` returns the list of calls that involved the object as a parameter, but were not nested in any other call that involved the object as a parameter. `addNewNode` adds a new node to the graph. `addEdge` adds an edge for the method given by its first parameter, between the nodes given by its second and third parameters. `getNestedCalls` returns a call sequence containing all the calls nested in its parameter. `replaceNode` replaces, in the graph, the node pointed to by its first parameter with the node pointed to by its second.

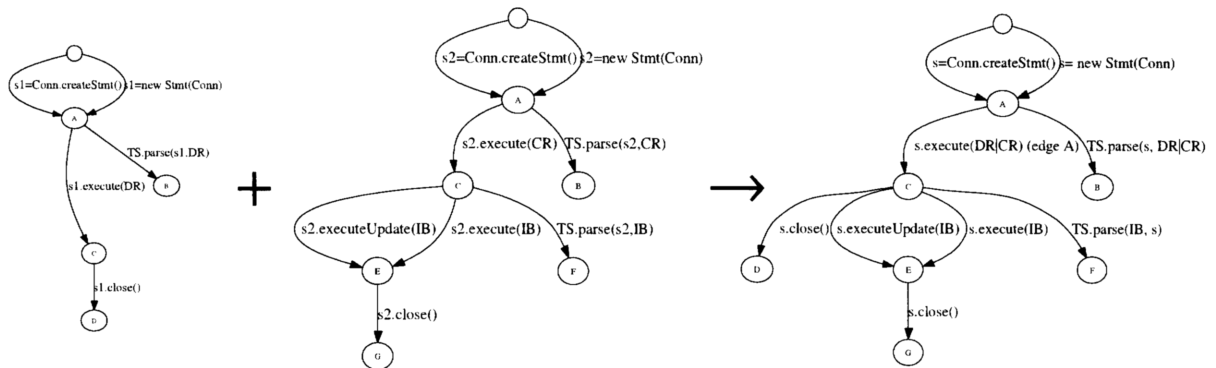


Figure 2-4: Call sequence graphs for $s1$ (from Figure 2-2(c)), $s2$ (not presented elsewhere), and the merged graph for class `Statement`.

2.1.1.2 Merging Call Sequence Graphs

Once the algorithm finished generating the call sequence graphs for all the observed instances of each class, the algorithm merges them into the class's model as follows. First, merge their root nodes. Whenever two nodes are merged, merge any pair of outgoing edges (and their target nodes) if (i) the edges record the same method, and (ii) the object appears in the same parameter positions (if the object is the receiver of the first method it must be the receiver of the second, similarly for the parameters); other parameters, including primitives and strings may differ. When two edges are merged, the new edge stores their combined set of primitives and strings.

Figure 2-4 shows merging of call sequence graphs. The left and center parts show the graphs for

s1 and s2, while the right part shows the merged model for the `Stmt` class. The edges corresponding to `s1.execute`(DR) and `s2.execute`(CR) are merged to create the edge `s.execute`(DR|CR).

2.1.1.3 Advantages of the Call Sequence Graph

Following the call sequence graph can lead to generation of legal sequences that were not seen in the original execution, or that exercise the program behavior in interesting corner cases. For instance, a valid call sequence from the model in Figure 2-4(c) might try to create a table and then close the connection (in the original execution, after the table was created, the statement was used to issue a query). Another possible sequence might try to drop the table "test" and then try to insert rows into it, causing a missing table exception to be thrown. Both of those examples exhibit behavior that should be checked by a regression test. For instance, the regression test for the latter would check for the missing table exception.

2.1.2 Generating Test Inputs

The input generator uses the inferred call sequence models to guide generation towards legal sequences. The generator has three arguments: (1) a set of classes for which to generate inputs, (2) call sequence models for a subset of the classes (those for which the user wants test inputs generated using the models), and (3) a time limit. The result of the generation is a set of test inputs for the classes under test.

The input generator mixes generation using universal models (that allow any method sequence and any parameters) and generation using the models inferred from the example execution. The generator is incremental: it maintains an (initially empty) *component set* of previously-generated method sequences, and creates new sequences by extending sequences from the component set with new method calls.

The input generation has two phases, each using a specified fraction of the overall time limit. In the first phase, the generator uses universal models to create test inputs. The purpose of this phase is initializing the component set with sequences that can be used during learned model-based generation. This phase may create sequences that do not follow the inferred models, which allows for creation of more diverse test inputs. In the second phase, the generator uses the inferred models to guide the creation of new test inputs.

An important challenge in our approach is creating tests that differ sufficiently from observed executions. Our technique achieves this goal by (i) generalizing observed sequences (inferred models may contain paths not observed during execution), (ii) omitting certain details from models (e.g., values of non-primitive, non-string parameters) (iii) diverging from models by randomly inserting calls to methods not observed during execution, and (iv) sequences that are created in the first phase of generation can be inserted in the second phase.

2.1.2.1 Phase 1: Generation Using Universal Models

In this phase, the generator executes the following three steps in a loop, until the time limit expires.

1. **Select a method.** Select a method $m(T_0, \ldots, T_K)$ at random from among the public methods declared in the classes under test (T_0 is the type of the receiver). The new sequence will have this method as its last call.

2. **Create a new sequence.** For type T_i of each parameter of method m, attempt to find, in the component set, an argument of type T_i for method m. The argument may be either a primitive value or a sequence s_i that creates a value of type T_i. There are two cases:

   - If T_i is a primitive (or string) type, then select a primitive value at random from a pool of primitive inputs (our implementation seeds the pool with inputs like `0`, `1`, `-1`, `'a'`, `true`, `false`, "", etc.).
   - If T_i is a reference type, then use `null` as the argument, or select a random sequence s_i in the component set that creates a value of type T_i, and use that value as the argument. If no such sequence exists, go back to step 1.

   Create a new sequence by concatenating the s_i sequences and appending the call of m (with the chosen parameters) to the end.

3. **Add the sequence to the component set.** Execute the new sequence (our implementation uses reflection to execute sequences). If executing the sequence does not throw an exception, then add the sequence to the component set. Otherwise, discard the sequence. Sequences that throw exceptions are not useful for further input generation. For example, if the one-method input `a = sqrt(-1);` throws an exception because the input argument must be non-negative, then there is no sense in building upon it to create the two-method input `a = sqrt(-1); b = log(a);`.

Example. We illustrate input generation using the universal models on the `tinySQL` classes. In this example, the generator creates test inputs for classes `Driver` and `Conn`. In the first iteration, the generator selects the static method `Conn.create(Stmt)`. There are no sequences in the component set that create a value of type `Stmt`, so the generator goes back to step 1. In the second iteration, the generator selects the constructor `Driver()` and creates the sequence `Driver d = new Driver()`. The generator executes the sequence, which throws no exceptions. The generator adds the sequence to the component set. In the third iteration, the generator selects the method `Driver.connect(String)`. This method requires two arguments: the receiver of type `Driver` and the argument of type `String`. For the receiver, the generator uses the sequence `Driver d = new Driver();` from the component set. For the argument, the generator randomly selects "" from the pool of primitives. The sequence is `Driver d = new Driver(); d.connect("")`. The generator executes the sequence, which throws an exception (i.e., the string "" is not valid a valid argument). The generator discards the sequence.

2.1.2.2 Phase 2: Generation Using the Inferred Models

In this phase the generator uses the inferred model to guide the creation of new sequences. We call the sequences that the model-based generator creates *modeled sequences*, which are distinct from

the sequences generated in the first phase. The generator keeps two (initially empty) mappings. Once established, the mappings never change for a given modeled sequence. The *mo* (modeled object) mapping maps each modeled sequence to the object constructed by the sequence. The *cn* (current node) mapping maps each modeled sequence to the node in the model that represents the current state of the sequence's *mo*-mapped object.

Similarly to the generator from Phase 1, the generator attempts to create a new sequences by repeatedly extending modeled sequences from the component set. The component set is initially populated with the (universally modeled) sequences created in the first generation phase. The generator repeatedly performs one of the following two actions (randomly selected), until the time limit expires.

- **Action 1: create a new modeled sequence.** Select at random a class C and an edge E that is outgoing from the root node in the model of C Let $m(T_0, \ldots, T_k)$ be the method that edge E represents. Create a new sequence s' that ends with a call to m, in the same manner as the generation in phase 1—concatenate any sequences from the component set to create the arguments for the call, then append the call to m at the end. Execute s' and add it to the component set if it terminates without throwing an exception. Create the *mo* mapping for s'—the s' sequence *mo*-maps to the return value of the call to m (model inference ensures that m does have a return value). Finally, create the initial *cn* mapping for s'—the s' sequence *cn*-maps to the target node of the E edge.

- **Action 2: extend an existing modeled sequence.** Select a *modeled* sequence s from the component set and an edge E outgoing from the node $cn(s)$ (i.e., from the node to which s maps by *cn*). These selections are done at random. Create a new sequence s' by extending s with a call to the method that edge E represents (analogously to Action 1). If a parameter of m is of a primitive or string type, randomly select a value from among those that decorate edge E. Execute s' and add it to the component set if it terminates without throwing an exception. Create the *mo* mapping for s'—the s' sequence *mo*-maps to the same value as sequence s. This means that s' models an object of the same type as s. Finally, create the *cn* mapping for s'—the s' sequence *cn*-maps to the target node of the E edge.

Example. We use `tinySQL` classes to show an example of how the generator works. The generator in this example uses the model presented in the right-hand side of Figure 2-4. In the first iteration, the generator selects Action 1, and method `createStmt`. The method requires a receiver, and the generator finds one in the component set populated in first generation phase (last two lines of the following example) The method executes with no exception thrown and the generator adds it to the component set. The following shows the newly created sequence together with the *mo* and *cn* mappings.

sequence s	$mo(s)$	$cn(s)$

`Driver d = new Driver();`
`Conn c = d.connect("jdbc:tinySQL");`
`Statement st = c.createStmt();`
	st	A

In the second iteration, the generator selects Action 2 and method `execute`. The method requires a string parameter and the model is decorated with two values for this call (denoted by DR and CR in the right-most graph of Figure 2-4). The generator randomly selects CR. The method executes with no exception thrown and the generator adds it to the component set. The following shows the newly created sequence together with the *mo* and *cn* mappings.

sequence s	*mo(s)*	*cn(s)*

`Driver d = new Driver();`
`Conn c = d.connect("jdbc:tinySQL");`
`Statement st = c.createStmt();`
`st.execute("CREATE TABLE test (name char(25), id int)");`
	st	C

2.2 Evaluation

This section presents an empirical evaluation of Palulu's ability to create test inputs. We mainly compared Palulu to JOE [111]. JOE is a test input generation technique that uses universal models, and was shown to out perform all of its competitors. Section 2.2.1 shows that test inputs generated using Palulu inferred models yields better coverage than test inputs generated by JOE. Section 2.2.2 shows that Palulu regression tests can find more regression errors than regression tests generated by JOE, and a hand crafted test suite. Section 2.2.3 illustrates that Palulu can create a test input for a complex data structure, for which JOE was unable to create a test input.

2.2.1 Coverage

We compared Palulu generation's using the inferred models to JOE [111]'s generation using the universal models in creating inputs for programs with constrained APIs. Our hypothesis is that tests generated by following the call sequence models will be more effective, since the test generator is able to follow method sequences and use input arguments that emulate those seen in an example input. We measure effectiveness via block and class coverage, since a test suite with greater coverage is generally believed to find more errors.

2.2.1.1 Subject programs

We used four Java programs as our subject programs. Each of these programs contains many classes with constrained APIs, requiring specific method calls and input arguments to create legal input.

- **tinySQL**[3] (27 kLOC) is a minimal SQL engine. We used the program's test suite as an example input.

[3] `http://sourceforge.net/projects/tinysql`

Program	all classes	tested classes	classes for which technique generated at least one input		block coverage	
			JOE	Palulu	JOE	Palulu
tinySQL	119	32	19	30	19%	32%
HTMLParser	158	22	22	22	34%	38%
SAT4J	122	22	22	22	27%	36%
Eclipse	320	70	46	46	8.0%	8.5%

Figure 2-5: Classes for which inputs were successfully created, and coverage achieved, by using JOE's test input generation models, and Palulu's model-based generation.

- **HTMLParser**[4] (51 kLOC) is real-time parser for HTML. We used our research group's webpage as an example input.
- **SAT4J**[5] (11 kLOC) is a SAT solver. We used a file with a non-satisfiable formula, taken from DIMACS[6], as an example input.
- **Eclipse compiler**[7] (98 kLOC) is the Java compiler supplied with the Eclipse project. We wrote a 10-line program for the compiler to process, as an example input.

2.2.1.2 Methodology

As the set of classes to test, we selected all the public non-abstract classes, for which instances where created during the sample execution. For other classes, we do not infer models and therefore the inputs generated by the two techniques will be the same.

The test generation was run in two phases. In the first phase it generated components for 20 seconds using universal models for all the classes in the application. In the second phase, test input creation, it generated test inputs for 20 seconds for the classes under test using either Palulu's inferred models or the universal models.

Using the generated tests, we collected block and class coverage information with emma[8].

2.2.1.3 Results

Figure 2-5 shows the results. The test inputs created by following the call sequence models achieve better coverage than those created by JOE.

The class coverage results differ only for tinySQL. For example, without the call sequence models, a valid connection or a properly-initialized database are never constructed, because of the required initialization methods and specific input strings.

The block coverage improvements are modest for Eclipse (6%, representing 8.5/8.0) and HTML-Parser (12%). SAT4J shows a 33% improvement, and tinySQL, 68%. We speculate that programs

[4] `http://htmlparser.sourceforge.net`
[5] `http://www.sat4j.org`
[6] `ftp://dimacs.rutgers.edu`
[7] `http://www.eclipse.org`
[8] `http://emma.sourceforge.net`

Test Suite	Number of Tests	Faulty Implementation	False Positives
Manually-Written	190	14	0
JOE	2016	42	1
Palulu	1465	64	1

Figure 2-6: Faulty implementations found in the RatPoly experiment by three test suites. Manually-written, generated by Palulu, and generated by JOE [111]. The first column is the number of tests in the test suite. The second column contains the number of faulty implementations reported by the test suite. The third column is the number of correct implementation that failed at least one test (false positive).

with more constrained interfaces, or in which those interfaces play a more important role, are more amenable to the technique.

The results are not dependent on the particular time bound chosen. For example, allowing JOE 100 seconds for generation achieved less coverage than generation using the call sequence models for 10 seconds.

2.2.2 Palulu Generate Effective Regression Tests

This section describes an experiment that evaluates the effectiveness of Palulu's regression test generation. We compared the generation using Palulu models against JOE [111] which generates tests using the universal models, and against a manually-written test suite (crafted over six years). Briefly, a test suite generated using the Palulu models found more bugs than both the suite generated by JOE, and the manually-written suite.

2.2.2.1 Subject programs

RatPoly is a library for manipulating rational-coefficient polynomials. Implementing the required API was an assignment in MIT's software engineering class. We had access to 143 student implementations, the staff implementation, and a test suite used for grading. The staff test suite contains 190 tests and 264 dynamic assertions and been augmented over the period of 6 years by the course staff. We have no reason to believe that its actual completeness was below the average state-of-the-practice in the real world. The staff solution (naturally) passed all the tests.

2.2.2.2 Methodology

We used Palulu and JOE to create a regression test suite for the staff implementation. Our goal was to discover faulty implementation submitted by the students. As the example input to Palulu, we wrote a small program exercising different methods of the RatPoly library.

Class	Description	Requires
VarInfoName	Variable name	
VarInfo	Variable description	VarInfoName
PptSlice2	Two variables from a program point	VarInfo, PptTopLevel, Invariant
PptTopLevel	Program point	PptSlice2, VarInfo
LinearBinary	Linear invariant ($y = ax + b$) over two scalar variables	PptSlice2
BinaryCore	Helper class	LinearBinary

Figure 2-7: Some of the classes needed to create a valid test input for Daikon's BinaryCore class. For each class, the **requires** column contains the types of all objects one needs to construct an object of that class.

2.2.2.3 Results

Palulu found more faulty implementations than JOE, even though it contains fewer tests; both Palulu and Joe found more faulty implementations than the manual test suite.

Figure 2-6 compares the results for the manually-written test suite, Palulu, and JOE. We have manually inspected all failing test cases and found out that they have been triggered by 19 underlining errors.

2.2.3 Constructing a Complex Input

To evaluate the technique's ability to create structurally complex inputs, we applied it to the BinaryCore class within Daikon [59], a tool that infers program invariants. Daikon maintains a complex data structure involving many classes to keep track of the valid invariants at each program point. BinaryCore is a helper class that calculates whether or not the points passed to it form a line. BinaryCore was suggested by a developer of Daikon as an example of a class whose creation requires a nontrivial amount of setup. We now describe this setup in broad strokes to give the reader an idea of the complexity involved (see Figure 2-7 for a list of constraints):

- The constructor to a BinaryCore takes an argument of type Invariant, which has to be of run-time type LinearBinary or PairwiseLinearBinary, subclasses of Invariant. Daikon contains 299 classes that extend Invariant, so the state space of type-compatible but incorrect possibilities is very large.
- To create a legal LinearBinary, one must first create a legal PptTopLevel and a legal PptSlice2. Both of these classes require an array of VarInfo objects. The VarInfo objects passed to PptSlice2 must be a subset of those passed to PptTopLevel. In addition, the constructor for PptTopLevel requires a string in a specific format; in Daikon, this string is read from a line in the input file.
- The constructor to VarInfo takes five objects of different types. Similar to PptTopLevel, these objects come from constructors that take specially-formatted strings.
- None of the parameters involved in creating a BinaryCore or any of its helper classes may be null.

Manually-written test input (written by an expert)	Palulu-generated test input

`VarInfoName namex = VarInfoName.parse("x");`
`VarInfoName namey = VarInfoName.parse("y");`
`VarInfoName namez = VarInfoName.parse("z");`
	`VarInfoName name1 = VarInfoName.parse("return");`
	`VarInfoName name2 = VarInfoName.parse("return");`

`ProglangType inttype = ProglangType.parse("int");`
`ProglangType filereptype = ProglangType.parse("int");`
`ProglangType reptype = filereptype.fileToRepType();`
	`ProglangType type1 = ProglangType.parse("int");`
	`ProglangType type2 = ProglangType.parse("int");`

`VarInfoAux aux = VarInfoAux.parse("");`	`VarInfoAux aux1 =`
	` VarInfoAux.parse(" declaringClassPackageName=, ");`
	`VarInfoAux aux2 =`
	` VarInfoAux.parse(" declaringClassPackageName=, ");`

`VarComparability comp =`
`VarComparability.parse(0, "22", inttype);`
	`VarComparability comp1 =`
	` VarComparability.parse(0, "22", type1);`
	`VarComparability comp2 =`
	` VarComparability.parse(0, "22", type2);`

`VarInfo v1 =`
` new VarInfo(namex, inttype, reptype, comp, aux);`
`VarInfo v2 =`
` new VarInfo(namey, inttype, reptype, comp, aux);`
`VarInfo v3 =`
` new VarInfo(namez, inttype, reptype, comp, aux);`
	`VarInfo v1 =`
	` new VarInfo(name1, type1, type1, comp1, aux1);`
	`VarInfo v2 =`
	` new VarInfo(name2, type2, type2, comp2, aux2);`

`VarInfo[] slicevis = new VarInfo[] {v1, v2};` `VarInfo[] pptvis = new VarInfo[] {v1, v2, v3};`	`VarInfo[] vs = new VarInfo[] {v1, v2};`

`PptTopLevel ppt =`
` new PptTopLevel("StackAr.StackAr(int):::EXIT33",`
` pptvis);`
	`PptTopLevel ppt1 =`
	` new PptTopLevel("StackAr.push(Object):::EXIT", vs);`

`PptSlice2 slice = new PptSlice2(ppt, slicevis);`	`PptSlice slice1 = ppt1.gettempslice(v1, v2);`
`Invariant proto = LinearBinary.getproto();` `Invariant inv = proto.instantiate(slice);`	`Invariant inv1 = LinearBinary.getproto();` `Invariant inv2 = inv1.instantiate(slice1);`
`BinaryCore core = new BinaryCore(inv);`	`BinaryCore lbc1 = new BinaryCore(inv2);`

Figure 2-8: The code listing on the left is a test input written by an expert developer of Daikon. It required about 30 minutes to write. The code listing on the right is a test input generated by Palulu using inferred models created from an example execution of Daikon. For ease of comparison, we renamed automatically-generated variable names and grouped method calls related to each class (but we preserved any ordering that affects the results).

First, we tested our hypothesis that generating a legal instance of `BinaryCore` is not trivial. JOE [111] was unable to generate a legal instance of `BinaryCore` in 24 hours. Next, we created the model, using as input to Daikon an example supplied with the Daikon distribution. Our input generator generated 3 sequences that create legal different `BinaryCore` instances, and about 150 helper sequences in 10 seconds.

Figure 2-8 (left) shows a test input that creates a legal `BinaryCore` instance. This test was written by a Daikon developer, who spent about 30 minutes writing the test input. We are not aware of a simpler way to obtain a `BinaryCore`.

Figure 2-8 (right) shows one of the three inputs that Palulu generated for `BinaryCore`. For ease of comparison between the inputs generated manually and automatically, we renamed automatically-named variables and reordered method calls when the reordering did not affect the results. Palulu successfully generated all the helper classes involved. Palulu generated some objects in a different way from the manual input; for example, to generate a `Slice`, Palulu used the return value of a method in `PptTopLevel` instead of the class's constructor.

2.3 Related Work

Palulu combines dynamic call sequence graph inference with test input generation. Previous techniques for test input generation required hand written models, while techniques that used dynamic analysis to generate models focused on program understanding. This section discusses related work in each area in more detail.

2.3.1 Dynamic Call Sequence Graph Inference

There is a large literature on call sequence graph inference; we discuss some techniques most closely related to our work. Cook and Wolf [36] generate a FSM from a linear trace of atomic, parameter-less events using grammar-inference algorithms [9]. Whaley and Lam [154] combine dynamic analysis of a program run and static analysis of the program's source to infer pairs of methods that cannot be called consecutively. Meghani and Ernst [97] improve on Whaley's technique by dynamically inferring methods specifications, and using them to create FSMs that are logically consistent with the specifications. Ammons et al. [7] use machine learning to generate the graph; like our technique, Ammon's is inexact (i.e., the inferred state machine allows more behaviors than those observed in the trace).

In all the above techniques, the intended consumer of the inferred graphs is a person wanting to gain program understanding. Our end goal is generating test inputs for object-oriented APIs; the consumer of our graphs is a mechanical test input generator, and the model is only as good as it is helpful in generating inputs. This fact imposes special requirements that our inference technique addresses. To be useful for real programs, our call sequence graph inference technique must handle program traces that include methods with multiple input parameters, nested calls, private calls, primitive parameters, etc. On the other hand, the size of the graph is less crucial to us. In addition, the models of the above techniques mostly discover rules affecting one object (for

instance, opening a connection before using it). In contrast, our model inference discovers rules consisting of many objects and method calls.

Another related project is Perracotta [161], which dynamically infers temporal properties from traces, such as "event E_1 always happens before E_2." Our call sequence graphs encode specific event sequences, but do not generalize the observations. Using inferred temporal properties could provide even more guidance to a test input generator.

2.3.2 Generating Test Inputs with a Model

A large body of existing work addresses the problem of generating test inputs from a specification or model; below we survey the most relevant.

Most of the previous work on generating inputs from a specification of legal method sequences [29, 35, 52, 70, 73, 76, 108, 121, 144, 147] expects the user to write the specification by hand, and assumes that all inputs derived from the specification are legal. In addition, many of these techniques are designed primarily for testing reactive systems or single classes such as linked lists, stacks, etc. whose methods typically can take any objects as parameters. This greatly simplifies input generation—there are fewer decisions to make, such as how to create an object to pass as a parameter.

Like Palulu, the Agedis [73] and Jartege [108] tools use randomization during the test input generation; Agedis requires the user to write a model as a UML diagram, and Jartege requires the user to provide JML specifications. The tools can generate inputs based on the models; the user also provides an oracle to determine whether an input is legal, and whether it is fault-revealing. Compared to Palulu, these tools represent a different trade-off in user control versus automation.

Since we use an automatically-generated model and apply our technique to realistic programs, our test input generator must account for any lack of information in the generated model and still be able to generate inputs for data structures. Randomization helps here: whenever the generator faces a decision (typically due to under-specification in the generated model), a random choice is made. As our evaluation shows, the randomized approach leads to legal inputs. Of course, this process can also lead to creation of illegal structures.

An alternative approach to creating objects is via direct heap manipulation (e.g., Korat [20]). Instead of using the public interface of an object's class, Korat constructs an object by directly setting values of the object's fields (public and private). To ensure that this approach produces legal objects, the user provides a detailed object invariant specifying legal objects. Our approach does not require a manually-written invariant to create test inputs. Instead, it infers a model and uses it to guide the random search towards legal object states.

2.3.3 Creating Unit Tests from System Tests

Our technique captures exhibited behavior at the unit level. It can be used to create a set of unit tests from a large (potentially long-running) system test. This process can be viewed as a refactoring of the system test into a set of smaller unit tests.

Test factoring [109, 129] is a different technique to create unit tests from a system test. Test factoring captures the interactions between tested and untested components in the system, and

creates mock objects for the untested portions of the interaction, yielding a unit test where the environment is simulated via mocks. Test factoring accurately reproduces the execution of the entire system test. Our technique, on the other hand, uses the system test as only as a guide to create new sequences of method calls. Both techniques share the goal of creating focused, fast unit tests where there were none.

2.4 Conclusion

In this chapter, we have presented a test generation technique that helps developers prevent regression faults. The technique uses dynamic analysis to automatically generates unit-level regression test with structurally complex inputs for object oriented programs.

Our technique combines dynamic model inference with model-based test input generation to create high-quality test suites. The technique is targeted for programs that define constrained APIs for which generation with universal models alone cannot generate useful tests that satisfy the constraints. It guides generation with a model that summarizes method sequencing and method input constraints seen in an example execution.

We have implemented our technique for Java. Our experimental results show that test suites generated by our tool achieve better coverage than generation using the universal models. Our technique is capable of creating legal tests for data structures that take a human significant effort to test.

Chapter 3

Detection

This chapter presents a test generation technique to help tester in the challenge of automatically detecting and localizing failures in dynamic web applications. Dynamic test generation tools, such as DART [67], Cute [133], and EXE [27], generate tests by executing an application on concrete input values, and then creating additional input values by solving symbolic constraints derived from exercised control flow paths. To date, such approaches have not been practical in the domain of web applications, which pose special challenges due to the dynamism of the programming languages, the use of implicit input parameters, the use of persistent state, and the complex patterns of user interaction.

This chapter extends dynamic test generation to the domain of web applications that dynamically create web (HTML) pages during execution, which are typically presented to the user in a browser. We apply these techniques in the context of the scripting language PHP, one of the most popular languages for web programming. According to the internet research service, Netcraft[1], PHP powered 21 million domains as of April 2007, including large, well-known websites such as Wikipedia and WordPress.

Our goal is to find two kinds of failures in web applications: *execution failures* that are manifested as crashes or warnings during program execution, and *HTML failures* that occur when the application generates malformed HTML. As an example, execution failures may occur when a web application calls an undefined function or reads a nonexistent file. In such cases, the HTML output contains an error message and execution of the application may be halted, depending on the severity of the failure. HTML failures occur when output is generated that is not syntactically well-formed HTML (e.g., when an opening tag is not accompanied by a matching closing tag). Although web browsers are designed to tolerate some degree of malformedness in HTML, several kinds of problems may occur. First and most serious is that browsers' attempts to compensate for malformed web pages may lead to crashes and security vulnerabilities[2]. Second, standard HTML renders faster[3]. Third, malformed HTML is less portable across browsers and is vulnerable to

[1] See `http://news.netcraft.com/`.

[2] See bug reports 269095, 320459, and 328937 at `https://bugzilla.mozilla.org/show_bug.cgi?`

[3] See `http://weblogs.mozillazine.org/hyatt/archives/2003_03.html#002904`. According to a Mozilla developer, one reason why malformed HTML renders slower is that "improper tag nesting [...] triggers residual style handling to try

breaking or looking strange when displayed by browser versions on which it is not tested. Fourth, a browser might succeed in displaying only part of a malformed webpage, while silently discarding important information. Fifth, search engines may have trouble indexing malformed pages [165].

Web developers widely recognize the importance of creating legal HTML. Many websites are checked using HTML validators[4]. However, HTML validators can only point out problems in HTML pages, and are by themselves incapable of finding faults in applications that *generate* HTML pages. Checking *dynamic* web applications (i.e., applications that generate pages during execution) requires checking that the application creates a valid HTML page on *every* possible execution path. In practice, even professionally developed and thoroughly tested applications often contain multiple faults (see Section 3.5).

There are two general approaches to finding faults in web applications: static analysis and dynamic analysis (testing). In the context of web applications, static approaches have limited potential because (i) web applications are often written in dynamic scripting languages that enable on-the-fly creation of code, and (ii) control in a web application typically flows via the generated HTML text (e.g., buttons and menus that require user interaction to execute), rather than solely via the analyzed code. Both of these issues pose significant challenges to approaches based on static analysis. Testing of dynamic web applications is also challenging, because the input space is large and applications typically require multiple user interactions. The state-of-the-practice in validation for web-standard compliance of real web applications involves the use of programs such as HTML Kit[5] that validate each generated page, but require manual generation of inputs that lead to displaying different pages. We know of no automated tool for the validation of web applications that dynamically generate HTML pages.

This chapter presents an automated technique for finding failures in dynamic web applications. Our technique is based on dynamic test generation, using combined concrete and symbolic (concolic) execution and constraint solving [27, 67, 133]. We created a tool, Apollo, that implements our technique in the context of the publicly available PHP interpreter.

Apollo first executes the web application under test with an empty input. During each execution, Apollo monitors the program to record the dependence of control-flow on input. Additionally, for each execution Apollo determines whether execution failures or HTML failures occur (for HTML failures, an HTML validator is used as an oracle). Apollo automatically and iteratively creates new inputs using the recorded dependence to create inputs that exercise different control flow. Most previous approaches for concolic execution only detect "standard errors" such as crashes and assertion failures. Our approach also detects such standard errors, but is to our knowledge the first to use an oracle to detect specification violations in the application's output.

Another novelty in our work is the inference of input parameters, which are not manifested in the source code, but which are interactively supplied by the user (e.g., by clicking buttons in generated HTML pages).

The desired behavior of a PHP application is usually achieved by a series of interactions between the user and the server (e.g., a minimum of five user actions are needed from opening the

to produce the expected visual result, which can be very expensive" [106].

[4]`http://validator.w3.org`, `http://www.htmlhelp.com/tools/validator`

[5]`http://www.htmlkit.com`

```
1  <?php
2
3  make_header(); // print HTML header
4
5  // Make the $page variable easy to use //
6  if(!isset($_GET['page'])) $page = 0;
7  else $page = $_GET['page'];
8
9  // Bring up the report cards and stop processing //
10 if($_GET['page2']==1337) {
11   require('printReportCards.php');
12   die();  // terminate the PHP program
13 }
14
15 // Validate and log the user into the system //
16 if($_GET["login"] == 1) validateLogin();
17
18 switch ($page)
19 {
20   case 0:  require('login.php'); break;
21   case 1:  require('TeacherMain.php'); break;
22   case 2:  require('StudentMain.php'); break;
23   default: die("Incorrect page number.  Please verify.");
24 }
25
26 make_footer(); // print HTML footer
27 ...
28 function validateLogin() {
29   if(!isset($_GET['username'])) {
30     echo "<j2> username must be supplied.</h2>\n";
31     return;
32   }
33   $username = $_GET['username'];
34   $password = $_GET['password'];
35   if($username=="john" && $password=="theTeacher")
36     $page=1;
37   else if($username=="john" && $password=="theStudent")
38     $page=2;
39   else echo "<h2>Login error. Please try again</h2>\n";
40 }
41
42 function make_header() { // print HTML header
43 print("
44 <!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01//EN"
45    "http://www.w3.org/TR/html4/strict.dtd">
46 <HTML>
47  <HEAD> <TITLE> Class Management  </TITLE> </HEAD>
48  <BODY>");
49 }
50
51 function make_footer() {  // close HTML elements opened by header()
52 print("
53  </BODY>
54 </HTML>");
55 }
56 ?>
```

Figure 3-1: A simplified PHP program excerpt from SchoolMate. This excerpt contains three faults (2 real, 1 seeded), explained in Section 3.1.3.

main Amazon page to buying a book). We handle this problem by enhancing the combined concrete and symbolic execution technique with explicit-state model checking based on automatic dynamic simulation of user interactions. In order to simulate user interaction, Apollo stores the state of the environment (database, sessions, cookies) after each execution, analyzes the output of the execution to detect the possible user options that are available, and restores the environment state before executing a new script based on a detected user option.

Techniques based on combined concrete and symbolic executions [27,67,133] may create multiple inputs that expose the same fault. In contrast to previous techniques, to avoid overwhelming the developer, our technique automatically identifies the minimal part of the input that is responsible for triggering the failure. This step is similar in spirit to Delta Debugging [34, 163]. However, since Delta Debugging is a general, *black-box* input minimization technique, it is oblivious to the properties of inputs. In contrast, our technique is *white-box*: it uses the information that certain inputs induce partially overlapping control flow paths. By intersecting these paths, our technique minimizes the constraints on the inputs within fewer program runs.

The contributions presented in this chapter are:

- We adapt the established technique of dynamic test generation, based on combined concrete and symbolic execution [27,67,133], to the domain of PHP web applications. This involved the following innovations: (i) using an HTML verifier as an oracle, (ii) inferring input parameters that are not manifested in the source code, (iii) dealing with datatypes and operations specific to the PHP language, (iv) tracking the use of persistent state and how input flows through it, and (v) simulating user input for interactive applications.

- We created a tool, Apollo, that implements the technique for PHP.

- We evaluated our tool by applying it to 6 real web applications and comparing the results with other tools. We show that dynamic test generation can be effective when adapted to the domain of web applications written in PHP: Apollo identified 302 faults while achieving line coverage of 50.2%.

- We present a detailed classification of the faults found by Apollo.

The remainder of this chapter is organized as follows. Section 3.1 presents an overview of PHP, introduces our running example, and discusses classes of failures in PHP web applications. Section 3.2 presents a simplified version of the algorithm and illustrates it on the example program. Section 3.3 presents the complete algorithm handling stateful execution with the simulation of interactive user inputs. Section 3.4 discusses our Apollo implementation. Section 3.5 presents our experimental evaluation of Apollo on open-source web applications. Section 3.6 gives an overview of related work, and Section 3.7 presents the chapter conclusions.

3.1 Context: PHP Web Applications

3.1.1 The PHP Scripting Language

This section briefly reviews the PHP scripting language, focusing on those aspects of PHP that differ from mainstream languages. Readers familiar with PHP may skip to the discussion of the running example in Section 3.1.2.

PHP is widely used for implementing web applications, in part due to its rich library support for network interaction, HTTP processing, and database access. The input to a PHP program is a map from strings to strings. Each key is a parameter that the program can read, write, or check if it is set. The string value corresponding to a key may be interpreted as a numerical value if appropriate. The output of a PHP web application is an HTML document that can be presented in a web browser.

PHP is object-oriented, in the sense that it has classes, interfaces, and dynamically dispatched methods with syntax and semantics similar to that of Java. PHP also has features of scripting languages, such as dynamic typing and an `eval` construct that interprets and executes a string value that was computed at run-time as a code fragment. For example, the following code fragment:

```
$code = "$x = 3;"; $x = 7; eval($code); echo $x;
```

prints the value 3 (names of PHP variables start with the `$` character). Other examples of the dynamic nature of PHP are a predicate that checks whether a variable has been defined, and class and function definitions that are statements that may occur anywhere.

The code in Figure 3-1 illustrates the flavor of PHP. The `require` statement that used on line 11 of Figure 3-1 resembles the C `#include` directive in the sense that it includes the code from another source file. However, the C version is a pre-processor directive with a constant argument, whereas the PHP version is an ordinary statement in which the file name is computed at run-time. There are many similar cases where run-time values are used, e.g., `switch` labels need not be constant. This degree of flexibility is prized by PHP developers for enabling rapid application prototyping and development. However, the flexibility can make the overall structure of program hard to discern and it can make programs prone to code quality problems.

3.1.2 PHP Example

The PHP program of Figure 3-1 is a simplified version of SchoolMate[6], which allows school administrators to manage classes and users, teachers to manage assignments and grades, and students to access their information.

Lines 6–7 read the global parameter `page` that is supplied to the program in the URL, e.g., `http://www.mywebsite.com/index.php?page=1`. Line 10 examines the value of the global parameter `page2` to determine whether to evaluate file `printReportCards.php`.

Function `validateLogin` (lines 28–40) sets the global parameter `page` to the correct value based on the identity of the user. This value is used in the `switch` statement on line 18, which presents the login screen or one of the teacher/student screens.

[6] `http://sourceforge.net/projects/schoolmate`

3.1.3 Failures in PHP Programs

Our technique targets two types of failures that can be automatically identified during the execution of PHP web applications. First, *execution failures* may be caused by a missing included file, an incorrect MySQL query, or by an uncaught exception. Such failures are easily identified as the PHP interpreter generates an error message and halts execution. Less serious execution failures, such as those caused by the use of deprecated language constructs, produce obtrusive error messages but do not halt execution. Second, *HTML failures* involve situations in which the generated HTML page is not syntactically correct according to an HTML validator. In the beginning of this chapter we discussed several negative consequences of malformed HTML.

As an example, the program of Figure 3-1 contains three faults, which cause the following failures when the program is executed:

1. Executing the program results in an *execution failure*: the file `printReportCards.php` referenced on line 11 is missing.

2. The program produces *malformed HTML* because the `make_footer` method is not executed in certain situations, resulting in an unclosed HTML tag in the output. In particular, the default case of the `switch` statement on line 23 terminates program execution when the global parameter `page` is not 0, 1, or 2 and when `page` is not written by function `ValidateLogin`.

3. The program produces *malformed HTML* when line 30 generates an illegal HTML tag `j2`.

The first failure is similar to a failure that our tool found in one of the PHP applications we studied. The second failure is caused by a fault that exists in the original code of the SchoolMate program. The third failure is the result of a fault that was artificially inserted into the example for illustration.

3.2 Finding Failures in PHP Web Applications

Our technique for finding failures in PHP applications is a variation on an established dynamic test generation technique [27,67,68,133] sometimes referred to as concolic testing. For expository purposes, we will present the algorithm in two steps. First, this section presents a simplified version of the algorithm that does not simulate user inputs or keep track of persistent session state. We will demonstrate this simplified algorithm on the example of Figure 3-1. Then, Section 3.3 presents a generalized version of the algorithm that handles user input simulation and stateful executions, and illustrates it on a more complex example.

The basic idea behind the technique is to execute an application on some initial input (e.g., an arbitrarily or randomly-chosen input), and then on additional inputs obtained by solving constraints derived from exercised control flow paths. We adapted this technique to PHP web applications as follows:

• We extend the technique to consider failures other than execution failures by using an oracle to determine whether or not program output is correct. In particular, we use an HTML validator to determine whether the output is a well-formed HTML page.

- The PHP language contains constructs such as `isset` (checking whether a variable is defined), `isempty` (checking whether a variable contains a value from a specific set), `require` (dynamic loading of additional code to be executed), `header` for redirection of execution, and several others that require the generation of constraints that are absent in languages such as C or Java.
- PHP applications typically interact with a database and need appropriate values for user authentication (i.e., user name and password). It is not possible to infer these values by either static or dynamic analysis, or by randomly guessing. Therefore, our technique uses a pre-specified set of values for database authentication. E.g., User=joe, Password="12345".

3.2.1 Algorithm

Figure 3-2 shows pseudo-code for our algorithm. The inputs to the algorithm are: a program $\mathcal{P}$, an oracle for the output O, and an initial state of the environment $\mathcal{S}_0$. The output of the algorithm is a set of bug reports $\mathcal{B}$ for the program $\mathcal{P}$, according to O. The report consists of a single failure, defined by the error message and the set of statements that is related to the failure. In addition, the report contains the set of all inputs under which the failure was exposed, and the set of all path constraints that lead to the inputs exposing the failure.

The algorithm uses a queue of configurations. Each configuration is a pair of a path constraint and an input. A *path constraint* is a conjunction of conditions on the program's input parameters. The queue is initialized with the empty path constraint and the empty input (line 3). The program is executed concretely on the input (line 6) and tested for failures by the oracle (line 7). Then, the path constraint and input for each detected failure are merged into the corresponding bug report (lines 7–8).

Next, the algorithm uses a subroutine, newConfigs, to find new configurations. First, the program is executed symbolically on the same input (line 15). The result of symbolic execution is a path constraint, $\bigwedge_{i=1}^{n} c_i$, that is satisfied by the path that was just executed from entry to exit of the whole program. The subroutine then creates new inputs by solving modified versions of the path constraint (lines 16–20), as follows. For each prefix of the path constraint, the algorithm negates the last conjunct (line 17). A solution, if it exists, to such an alternative path constraint corresponds to an input that will execute the program along a prefix of the original execution path, and then take the opposite branch, presumably covering new code. The algorithm uses a constraint solver to find a concrete input for each path constraint (line 18).

3.2.2 Example

Let us now consider how the algorithm of Figure 3-2 exposes the third fault in the example program of Figure 3-1.

Iteration 1. The first input to the program is the empty input, which is the result of solving the empty path constraint. During the execution of the program on the empty input, the condition on line 6 evaluates to `true`, and `page` is set to `0`. The condition on line 10 evaluates to `false`. The condition on line 16 evaluates to `false` because parameter `login` is not defined. The `switch` statement on line 18 selects the case on line 20 because `page` has the value of `0`. Execution

```
parameters: Program P, oracle O, Initial state S_0
result     : Bug reports B;
B : setOf(⟨failure, setOf(pathConstraint), setOf(input)⟩)
1  B := ∅;
2  toExplore := emptyQueue();
3  enqueue(toExplore, ⟨emptyPathConstraint(), emptyInput⟩);
4  while not empty(toExplore) and not timeExpired() do
5      ⟨pathConstraint, input⟩ := dequeue(toExplore);
6      output := executeConcrete(S_0, P, input);
7      foreach f in getFailures(O, output) do
8          merge ⟨f, pathConstraint, input⟩ into B;
9      newConfigs := getConfigs(input);
10     foreach ⟨pathConstraint_i, input_i⟩ ∈ newConfigs do
11         enqueue(toExplore, ⟨pathConstraint_i, input_i⟩);
12 return B;

13 Subroutine getConfigs(input):
14 configs := ∅;
15 c_1 ∧ ... ∧ c_n := executeSymbolic(S_0, P, input);
16 foreach i = 1,...,n do
17     newPC := c_1 ∧ ... ∧ c_{i-1} ∧ ¬c_i;
18     input := solve(newPC);
19     if input ≠ ⊥ then
20         enqueue(configs, ⟨newPC, input⟩);
21 return configs;
```

Figure 3-2: The failure detection algorithm. The output of the algorithm is a set of bug reports. Each bug report contains a failure, a set of path constraints exposing the failure, and a set of input exposing the failure. The *solve* auxiliary function uses the constraint solver to find an input satisfying the path constraint, or returns $\perp$ if no satisfying input exists. The *merge* auxiliary function merges the pair of pathConstraint and input for an already detected failure into the bug report for that failure.

terminates on line 26. The HTML verifier determines that the output is legal, and *executeSymbolic* produces the following path constraint:

$$NotSet(\texttt{page}) \wedge \texttt{page2} \neq \texttt{1337} \wedge \texttt{login} \neq \texttt{1} \quad \text{(I)}$$

The algorithm now enters the **foreach** loop on line 16 of Figure 3-2, and starts generating new path conditions by systematically traversing subsequences of the above path constraint, and negating the last conjunct. Hence, from (I), the algorithm derives the following three path constraints:

$$\begin{array}{ll} NotSet(\texttt{page}) \wedge \texttt{page2} \neq \texttt{1337} \wedge \texttt{login} = \texttt{1} & \text{(II)} \\ NotSet(\texttt{page}) \wedge \texttt{page2} = \texttt{1337} & \text{(III)} \\ Set(\texttt{page}) & \text{(IV)} \end{array}$$

Iteration 2. For path constraint (II), the constraint solver may find the following input (the solver is free to select any value for `page2`, other than 1337): `page2` $\leftarrow$ `0`, `login` $\leftarrow$ `1`.

When the program is executed with this input, the condition of the if-statement on line 16 evaluates to `true`, resulting in a call to the `validateLogin` method. Then, the condition of the if-statement on line 29 evaluates to `true` because the `username` parameter is not set, resulting in the generation of output containing an incorrect HTML tag `j2` on line 30. When the HTML validator checks the page, the failure is discovered and a bug report is created and added to the output set of bug reports.

3.2.3 Path Constraint Minimization

The failure detection algorithm (Figure 3-2) returns bug reports. Each bug report contains a set of path constraints, and a set of inputs exposing the failure. Previous dynamic test generation tools [27, 67, 133] presented the whole input (i.e., many ⟨*inputParameter*, *value*⟩ pairs) to the user without an indication of the subset of the input responsible for the failure. As a postmortem phase, our minimizing algorithm attempts to find a shorter path constraint for a given bug report (Figure 3-3). This eliminates irrelevant constraints, and a solution for a shorter path constraint is often a smaller input.

For a given bug report *b*, the algorithm first intersects all the path constraints exposing *b.failure* (line 1). The minimizer systematically removes one conjunct at a time (lines 3-6). If one of these shorter path constraints does not expose *b.failure*, then the removed conjunct is required for exposing *b.failure*. The set of all such required conjuncts determines the minimized path constraint. From the minimized path constraint, the algorithm produces a concrete input that exposes the failure.

The algorithm in Figure 3-3 does not guarantee that the returned path constraint is the shortest possible that exposes the failure. However, the algorithm is simple, fast, and effective in practice (see Section 3.5.3.2).

Our minimizer differs from *input* minimization techniques, such as delta debugging [34, 162], in that our algorithm operates on the *path constraint* that exposes the failure, and not the *input*. A constraint concisely describes a class of inputs (e.g., the constraint `page2` $\neq$ `1337` describes all

```
parameters: Program 𝒫, oracle O, bug report b
result    : Short path constraint that exposes b.failure
1  c_1 ∧ ... ∧ c_n := intersect(b.pathConstraints);
2  pc := true;
3  foreach i = 1, ..., n do
4      pc_i := c_1 ∧ ... c_{i-1} ∧ c_{i+1} ∧ ... c_n;
5      if !exposesFailures(pc_i) then
6          pc := pc ∧ c_i;
7  if exposesFailures(pc) then
8      return pc;
9  return shortest(b.pathConstraints);

10 Subroutine exposesFailure(pc):
11 input_pc := solve(pc);
12 if input_pc ≠ ⊥ then
13     output_pc := executeConcrete(𝒫, input_pc);
14     failures_pc := getFailures(O, output_pc);
15     return b.failure ∈ failures_pc;
16 return false;
```

Figure 3-3: The path constraint minimization algorithm. The method *intersect* returns the set of conjuncts that are present in all given path constraints, and the method *shortest* returns the path constraint with fewest conjuncts. The other auxiliary functions are the same as in Figure 3-2.

inputs different than 1337). Since a concrete input is an instantiation of a constraint, it is more effective to reason about input properties in terms of their constraints.

Each failure might be encountered along several execution paths that might partially overlap. Without any information about the properties of the inputs, delta debugging minimizes only a *single* input at a time, while our algorithm handles *multiple* path constraints that lead to a failure.

3.2.4 Minimization Example

The malformed HTML failure described in Section 3.2.2 can be triggered along different execution paths. For example, both of the following path constraints lead to inputs that expose the failure. Path constraint (a) is the same as (II) in Section 3.2.2.

$$\begin{array}{ll} \mathit{NotSet}(\texttt{page}) \land \texttt{page2} \neq 1337 \land \texttt{login} = 1 & (a) \\ \mathit{Set}(\texttt{page}) \land \texttt{page} = 0 \land \texttt{page2} \neq 1337 \land \texttt{login} = 1 & (b) \end{array}$$

First, the minimizer computes the intersection of the path constraints (line 1). The intersection is:

$$\texttt{page2} \neq 1337 \land \texttt{login} = 1 \quad (a \cap b)$$

Then, the minimizer creates two shorter path constraints by removing each of the two conjuncts in turn. First, the minimizer creates path constraint `login` = 1. This path constraint corresponds to an input that reproduces the failure, namely `login` ← 1. The minimizer determines this by executing the program on the input (line 14 in Figure 3-3). Second, the minimizer creates path constraint `page2` ≠ `1337`. This path constraint does not correspond to an input that exposes the failure. Thus, the minimizer concludes that the condition `login` = 1, that was removed from $(a \cap b)$ to form the second path constraint, is required. In this example, the minimizer returns `login` = 1. The result is the minimal path constraint that describes the minimal failure-inducing input, namely `login` ← 1.

3.3 Combined Concrete and Symbolic Execution with Explicit-State Model Checking

A typical PHP web application is a client-server application in which data and control flows interactively between a server that runs PHP scripts and a client, which is usually a web browser. The PHP scripts that run on the server generate HTML that includes interactive user input widgets such as buttons and menu items that, when selected by the user, invoke other PHP scripts. When these other PHP scripts are invoked, they are passed a combination of user input and constant values taken from the generated HTML. Modeling such user input is important, because coverage of the application will typically remain very low otherwise.

In Section 3.2, we described how to find failures in PHP web applications by adapting an existing test generation approach to consider language constructs that are specific to PHP, by using an oracle to validate the output, and by supporting database interaction. However, we did not yet supply a solution for handling user input options that are created dynamically by a web application, which includes keeping track of parameters that are transferred from one script to the next—either by persisting them in the environment, or by sending them as part of the call.

To handle this problem, Apollo implements a form of explicit-state software model checking. That is, Apollo systematically explores the state space of the system, i.e., the program under test. The algorithm in Section 3.2 always restarts the execution from the same initial state, and discards the state reached at the end of each execution. Thus, the algorithm reaches only 1-level deep into the application, where each level corresponds to a cycle of: a PHP script that generates an HTML form that the user interacts with to invoke the next PHP script. In contrast, the algorithm presented in this section remembers and restores the state between executions of PHP scripts. This technique, known as state matching, is widely known in model checking [78, 149] and implemented in tools such as SPIN [47] and JavaPathFinder [74]. To our knowledge, we are the first to implement state matching in the context of web applications and PHP.

3.3.1 Interactive User Simulation Example

Figure 3-4 shows an example of a PHP application that is designed to illustrate the particular

```
1  <html>
2  <head>Login</head>
3  <body>
4    <form name="login" action="exampleLogin.php">
5      <input type="text" name="user"/>
6      <input type="password" name="pw"/>
7    </form>
8  </body>
9  </html>
```

(a) index.php

```
10 <?php
11   userTag = 'user'
12   pwTag = 'pw';
13   typeTag = 'type';
14 ?>
```

(b) constants.php

```
15 <HTML>
16 <?php
17   require( dirname(__FILENAME__).
18            '/includes/constants.php');
19
20   $user = $_REQUEST[ 'user' ];
21   $pw = $_REQUEST[ 'pw' ];
22
23   if (check_password($user, $pw) {
24     print "<HEAD>Login Successful</HEAD>\n";
25
26   $_SESSION[ $userTag] = $user;
27   $_SESSION[ $pwTag ] = $pw;
28 ?>
29   <BODY>
30     <FORM action="view.php">
31       <INPUT TYPE="text" NAME="topic"/>
32     </FORM>
33   </BODY>
34 <?php
35   if ($user == 'admin') {
36     $_SESSION[ $typeTag ] = 'admin';
37   }
38   else {
39     print "<HEAD>Login Failed</HEAD>\n";
40   }
41 ?>
42 </HTML>
```

(c) login.php

```
43 <HTML>
44 <HEAD>Topic View</HEAD>
45 <?php
46   print "<BODY>\n";
47   if(check_password($_SESSION[$userTag], $_SESSION[$pwTag]) {
48     require( dirname(__FILENAME__).'/includes/constants.php');
49
50     $type = $_SESSION[ $typeTag ];
51     $topic = $_REQUEST[ 'topic' ];
52
53     if ($type == 'admin') {
54       print "<H1>Admin ";
55     } else {
56       print "<H1>Normal ";
57     }
58     print "View of $topic</H1>\n";
59
60     /* code to print topic view... */
61
62     if ($type == 'admin') {
63       print "<H2>Administrative Details\n";
64     /* code to print admin details... */
65     }
66   } else {
67       print "Please Log in\n";
68   }
69   print "</BODY>\n";
70 ?>
71 </HTML>
```

(d) view.php

Figure 3-4: Example of a PHP web application.

complexities of finding faults in an interactive web applications. In particular, the figure shows: an index.php top-level script that contains static HTML in Figure 3-4(a), a generic login script login.php in Figure 3-4(c), and a skeleton of a data display script view.php in Figure 3-4(d). The PHP scripts in Figure 3-4 rely on a shared include file constants.php that defines some standard constants, which is shown in in Figure 3-4(b). Note that the code in Figure 3-4 is an ad-hoc mixture of PHP statements and HTML fragments. The PHP code is delimited by <?php and ? > tokens (For instance lines 45 and 70 in Figure 3-4(c)). The use of HTML in the middle of a PHP indicates that HTML is generated as if it were the argument of a print statement. The dirname function—which returns the directory component of a filename—is used in the require statements, as an example of including a file whose name is computed at run-time.

These PHP scripts are part of the client-server work flow in a web application: the user first sees the index.php page of Figure 3-4(a) and enters credentials. The user-input credentials are processed by the script in Figure 3-4(c), which generates a response page that allows the user to enter further input—a topic—that in turn entails further processing by the script in Figure 3-4(d). Note that the user name and password that are entered by the user during the execution of login.php are stored in special locations $_SESSION[$userTag] and $_SESSION[$pwTag], respectively. Moreover, if the user is the administrator, this fact is recorded similarly, in $_SESSION[$typeTag]. These locations illustrate how PHP handles *session state*, which is data that persists from one page to another, typically for a particular interaction by a particular user. Thus, the updates to _SESSION in Figure 3-4(c) will be seen (as the SESSION information is saved and read locally on the server) by the code in Figure 3-4(d) when the user follows the link to view.php in the HTML page that is returned by login.php. The view.php script uses this session information to verify the username/password in line 47.

Our example program contains an error in the HTML produced for the administrative details: the H2 tag that is opened on line 63 of Figure 3-4(d) is not closed. While this fault itself is trivial, finding it is not. Assume that testing starts (as an ordinary user would) by entering credentials to the script in Figure 3-4(c). A tester must then discover that setting $user to the value 'admin' results in the selection of a different branch that records the user type 'admin' in the session state (see lines 35–37 in login.php). After that, a tester would have to enter a topic in the form generated by the login script, and would then proceed to Figure 3-4(d) with the appropriate session state, which will finally generate HTML exhibiting the fault as is shown in Figure 3-5(a). Thus, finding the fault requires a careful selection of inputs to a series of interactive scripts, as well as making sure that updates to the session state during the execution of these scripts are preserved (i.e., making sure that the execution of the different script happen during the same session).

3.3.2 Algorithm

Figure 3-6 shows pseudo-code for the algorithm, which extends the algorithm in Figure 3-2 with explicit-state model checking to handle the complexity of simulating user inputs. The algorithm tracks the state of the environment, and automatically discovers additional configurations based on an analysis of the output for available user options. In particular, the algorithm (i) tracks changes to the state of the environment (i.e., session state, cookies, and the database) and (ii) performs an "on the fly" analysis of the output produced by the program to determine what user options it contains,

```
1 <HTML>
2 <HEAD>Topic View</HEAD>
3 <BODY>
4 <H1>Admin View of A topic</H1>
 ...
5 <H2>Administrative Details
 ...
6 </BODY>
7 </HTML>
```

(a) HTML output

HTML line	*PHP lines in 3-4(d)*
1	43
2	44
3	46
4	54, 58
5	63
6	69
7	71

(b) output mapping

```
Error at line 6, character 7:  end tag for "H2" omitted; possible causes include a missing
end tag, improper nesting of elements, or use of an element where it is not allowed
Line 5, character 1:  start tag was here
```

(c) Output of WDG Validator

Figure 3-5: **(a)** HTML produced by the script of Figure 3-4(d). **(b)** Output mapping constructed during execution. **(c)** Part of output of WDG Validator on the HTML of Figure 3-5(a).

with their associated PHP scripts. By determining the state of the environment as it exists when an HTML page is produced, the algorithm can determine the environment in which additional scripts are executed as a result of user interaction. This is important because a script is much more likely to perform complex behavior when executed in the correct context (environment). For example, if the web application does not record in the environment that a user is logged in, most subsequent calls will terminate quickly (e.g., when the condition in line 47 of Figure 3-4(d) is false) and will not present useful information. For simplicity, the algorithm implicitly handles the fact that there are possibly multiple entry points into a PHP program. Thus, an input will contain the script to execute in addition to the values of the parameters. For instance, the first call might be to index.php script, while subsequent calls can execute other scripts.

There are four differences (underlined in the figure) with the simplified algorithm that was previously shown in Figure 3-2.

1. A configuration contains an explicit state of the environment (before the only state that was used was the initial state $\mathcal{S}_0$) in addition to the path constraint and the input (line 3).

2. Before the program is executed, the algorithm (method `executeConcrete`) will restore the environment to the state given in the configuration (line 7), and will return the new state of the environment after the execution.

3. When the `getConfigs` subroutine is executed to find new configurations, it analyzes the output (the actual mechanism of the analysis is explained and demonstrated in Section 3.4.3) to find new possible transitions from the new environment state (lines 24—26). Each transition is expressed as a pair of a path constraint and an input.

4. The algorithm uses a set of configurations that are already in the queue (line 14) and it performs state matching, in order to only explore new configurations (line 11).

```
parameters: Program 𝒫, oracle O, Initial state S_0
result      : Bug reports ℬ;
ℬ : setOf(⟨failure, setOf(pathConstraint), setOf(input)⟩)
1  ℬ := ∅;
2  toExplore := emptyQueue();
3  enqueue(toExplore, ⟨emptyPC(), emptyInput(), S_0⟩);
4  visited := {⟨emptyPathConstraint(), emptyInput(), S_0⟩};
5  while not empty(toExplore) and not timeExpired() do
6      ⟨pathConstraint, input, S_start⟩ := dequeue(toExplore);
7      ⟨output, S_end⟩ := executeConcrete(S_start, 𝒫, input);
8      foreach f in getFailures(O, output) do
9          merge ⟨f, pathConstraint, input⟩ into ℬ;
10     newConfigs := getConfigs(input, output, S_start, S_end);
11     newConfigs := newConfigs − visited;
12     foreach ⟨pathConstraint_i, input_i, S_i⟩ ∈ newConfigs do
13         enqueue(toExplore, ⟨pathConstraint_i, input_i, S_i⟩);
14         visited := visited ∪ ⟨S_i, input_i⟩;
15 return ℬ;

16 Subroutine getConfigs(input, output, S_start, S_end):
17 configs := ∅;
18 c_1 ∧ ... ∧ c_n := executeSymbolic(S_start, 𝒫, input);
19 foreach i = 1,...,n do
20     newPC := c_1 ∧ ... ∧ c_{i−1} ∧ ¬c_i;
21     input := solve(pathConstraint);
22     if input ≠ ⊥ then
23         enqueue(configs, ⟨newPC, input, S_start⟩);
24 foreach ⟨newInput_i, newPC_i⟩ ∈ analyzeOutput(output) do
25     if newInput ≠ ⊥ then
26         configs := configs ∪ ⟨newPC_i, newInput_i, S_end⟩;
27 return configs;
```

Figure 3-6: The failure detection algorithm. The output of the algorithm is a set of bug reports, each reports a failure and the set of tests exposing that failure. The *solve* auxiliary function uses the constraint solver to find an input satisfying the path constraint, or returns $\perp$ if no satisfying input exists. The *merge* auxiliary function merges the pair of pathConstraint and input for an already detected failure into the bug report for that failure. The *analyzeOutput* auxiliary function performs an analysis of the output to extract possible transitions from the current environment state.

3.3.3 Example

We will now illustrate the algorithm of Figure 3-6 using the example application of Figure 3-4. The inputs to the algorithm are: $\mathcal{P}$ is the code from Figure 3-4, the initial state of the environment is empty, the first script to execute is the script in Figure 3-4(a), and O is the WDG HTML validator[7]. The algorithm begins on line 3 by initializing the work queue with one item: an empty input to the script of Figure 3-4(a) with an empty path constraint and an empty initial environment.

iteration 1. The first iteration of the outer loop (lines 5–14) removes that item from the queue (line 6), restores the empty initial state, and executes the script (line 7).

No failures are observed. The call to *executeSymbolic* on line 18 returns an empty path constraint, so the function *analyzeOutput* on line 24 is executed next, and returns one user option; $\langle$`login`.php, $\emptyset$, $\emptyset\rangle$ for executing `login`.php with no input, and the empty state. This configuration is added to the queue (line 13) since it was not seen before.

iteration 2-5. The next iteration of the top-level loop dequeues the new work item, and executes `login`.php with empty input, and empty state. No failures are found. The call to *executeSymbolic* in line 18 returns a path constraint `user` $\neq$ `admin` $\wedge$ `user` $\neq$ `reg`, indicating that the call to `check_password` on line 23 in Figure 3-4(c) returned false[8]. Given this, the loop at lines 19–23 will generate several new work items for the same script with the following path constraints: `user` $\neq$ `admin`$\wedge$`user` = `reg`, and `user` = `admin` which are obtained by negating the previous path constraint. The loop on lines 24—26 is not entered, because no user input options are found. After several similar iterations, two inputs are discovered: `user` = `admin` $\wedge$ `pw` = `admin`, and `user` = `reg` $\wedge$ `pw` = `reg`. These corresponds to alternate control flows in which the `check_password` test succeeds.

iteration 6-7. The next iteration of the top-level loop dequeues an item. Using this item, the call to `check_password` will succeed (assume it selected `user` = `reg`...). Once again, no failures are observed, but now the session state with *user* and *pw* set is recorded at line 7. Also, this time *analyzeOutput* (line 24) finds the link to the script in Figure 3-4(d), and so the loop at lines 24—26 adds one item to the queue, executing `view`.php with the current session state.

The next iteration of the top-level loop dequeues one work item. Assume that it takes the last one described above. Thus, it executes the script in Figure 3-4(d) with a session that defines *user* and *pw* but not *type*. Hence, it produces an execution with no errors.

iteration 8-9. The next loop iteration takes that last work item, containing a user and password pair for which the call to `check_password` succeeds, with the user name as '`admin`'. Once again, no failures occur, but now the session state with *user*, *pw* and *type* set is recorded at line 7. This time, there are no new inputs to be derived from the path constraint, since all prefixes have been covered already. Once again, parsing the output finds the link to the script in Figure 3-4(d) and adds a work item to the queue, but with a different session state (in this case, the session state also

[7] http://htmlhelp.com/tools/validator/

[8] For simplicity, we omit the details of this function. It compares the user name and password to some constants '`admin`' and '`reg`'.

includes a value for *type*). The resulting execution of the script in Figure 3-4(d) with the session state that includes *type* results in an HTML failure.

3.4 Implementation

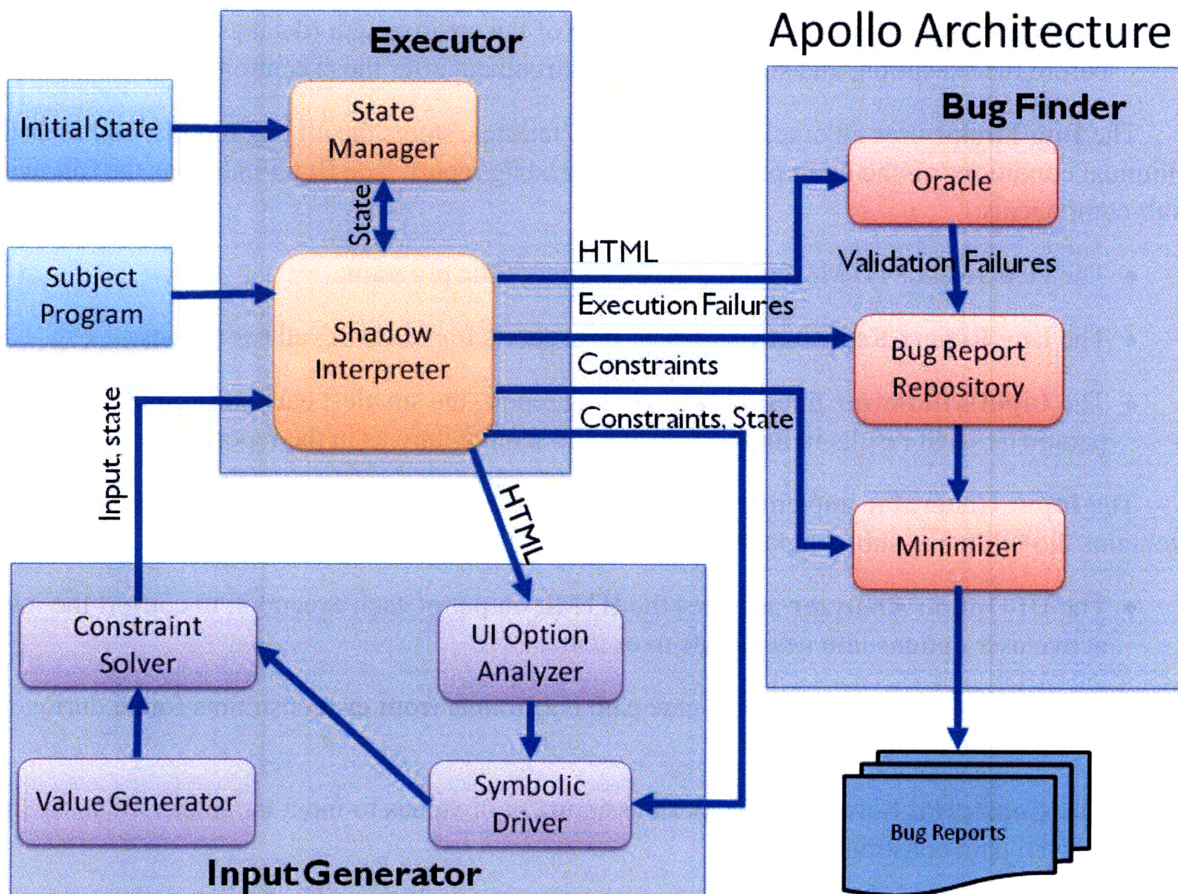


Figure 3-7: The architecture of Apollo.

We created a tool called Apollo that implements our technique for PHP. Apollo consists of three major components, **Executor**, **Bug Finder**, and **Input Generator** illustrated in Figure 3-7. This section first provides a high-level overview of the components and then discusses the pragmatics of the implementation.

The inputs to Apollo are the program under test and an initial value for the environment. The environment will usually consist of a database with some values, and additional information about username/password pairs for the database. Attempting to retrieve information from the database using randomly chosen values for username/password is unlikely to be successful. Symbolic execution is equally helpless without the database manager because reversing cryptographic functions

is beyond the state-of-the-art for constraint solvers.

The **Executor** is responsible for executing a PHP script with a given input in a given state. The executor contains two sub-components:

- The **Shadow Interpreter** is a PHP interpreter that we have modified to propagate and record path constraints and positional information associated with output. This positional information is used to determine which failures are likely to be symptoms of the same fault.
- The **State Manager** restores the given state of the environment (database, session, cookies) before the execution, and stores the new environment after the execution.

The **Bug Finder** uses an oracle to find HTML failures, stores the all bug reports, and finds the minimal conditions on the input parameters for each bug report. The Bug Finder has the following sub-components:

- The **Oracle** finds HTML failures in the output of the program.
- The **Bug Report Repository** stores all bug reports found during all executions.
- The **Input Minimizer** finds, for a given bug report, the smallest path constraint on the input parameters that results in inputs inducing the same failure as in the report.

The **Input Generator** implements the algorithm described in Figure 3-6. The Input Generator contains the following sub-components:

- The **UI Option Analyzer** analyzes the HTML output of each execution to convert the interactive user options into new inputs to execute.
- The **Symbolic Driver** generates new path constraints from the constraints found during the execution.
- The **Constraint Solver** computes an assignment of values to input parameters that satisfies a given path constraint.
- The **Value Generator** generates values for parameters that are not otherwise constrained, using a combination of random value generation and constant values mined from the program source code.

3.4.1 Executor

We modified the Zend PHP interpreter 5.2.2[9] to produce symbolic path constraints for the executed program, using the "shadow interpreter" approach [42]. The shadow interpreter performs the regular (concrete) program execution using the concrete values, and simultaneously performs symbolic execution. Creating the shadow interpreter required five alterations to the PHP run-time:

[9] `http://www.php.net/`

1. **Associating Symbolic Parameters with Values**

   A symbolic variable may be associated with each value. Values derived from the input—that is, either read directly as input or computed from input values—have symbolic variables associated with them. Values not derived from the input do not. These associations arise when a value is read from one of the special arrays _POST, _GET, and _REQUEST, which store parameters supplied to the PHP program. For example, executing the statement $x = $_GET["param1"] results in associating the value read from the global parameter param1 and bound to parameter x with the symbolic variable param1. Values maintain their associations through assignments and function calls (thus, the interpreter performs symbolic execution at the inter-procedural level). Importantly, during program execution, the concrete values remain, and the shadow interpreter does not influence execution.

   Unlike other projects that perform concrete and symbolic execution [27, 67, 68, 133], our interpreter does not associate complex symbolic expressions with all run-time values, but only symbolic variables, which exist only for input-derived values. This design keeps the constraint solver simple and reduces the performance overhead. As our results (Section 3.5) indicate, this lightweight approach is sufficient for the analyzed PHP programs.

2. **Storing Constraints at Branch Points**

   At branching points (i.e., value comparisons) that involve values associated with symbolic variables, the interpreter extends the initially empty path constraint with a conjunct that corresponds to the branch actually taken in the execution. For example, if the program executes a statement if($name == "John") and this condition succeeds, where $name is associated with the symbolic variable username, then the algorithm appends the conjunct username = "John" to the path constraint.

3. **Handling PHP Native Functions**

   Our modified interpreter records conditions for PHP-specific comparison operations, such as isset and empty, which can be applied to any variable. Operation isset returns a boolean value that indicates whether or not a value different from NULL was supplied for a variable. The empty operator returns true when applied to: the empty string, 0, "0", NULL, false, or an empty array. The interpreter records the use of isset on values with an associated symbolic variable, and on uninitialized parameters.

   The isset comparison creates either the *NotSet* or the *Set* condition. The constraint solver chooses an arbitrary value for a parameter p if the only condition for p is *Set* (p). Otherwise, it will also take into account other conditions. The *NotSet* condition is used only in checking the feasibility of a path constraint. A path constraint with the *NotSet* (p) condition is feasible only if it does not contain any other conditions on p. The empty comparison creates equality or inequality conditions between the parameter and the values that are considered empty by PHP.

4. **Propagating Inputs through Sessions and Cookies**

The use of session state allows a PHP application to store user-supplied information on the server for retrieval by other scripts. We enhanced the PHP interpreter to record when input parameters are stored in session state. This enables Apollo to track constraints on input parameters in all scripts that use them.

5. **Web Server Integration**

   Dynamic web applications often depend on information supplied by a web-server (such as Apache), and some PHP constructs are simply ignored by the command line interpreter (e.g., *header*). In order to allow Apollo to analyze more PHP code, Apollo supports execution through the Apache web-server in addition to the stand-alone command line executor. A developer can use Apollo to silently analyze the execution and record any failure found while manually using the subject program on an Apache server.

The modified interpreter performs symbolic execution along with concrete execution, i.e., every variable during program execution has a concrete value and may have additionally a symbolic value. Only the concrete values influence the control flow during the program execution, while the symbolic execution is only a "witness" that records, but does not influence, control flow decisions at branching points. This design deals with exceptions naturally because exceptions do not disrupt the symbolic-value mapping for variables.

Our approach to symbolic execution allows us to handle many PHP constructs that are problematic in a purely static approach. For instance, for computed variable names (e.g., `$x` = `${$foo}`), any symbolic information associated with the value that is held by the variable named by `foo` will be passed to `x` by the assignment[10]. In order to heuristically group HTML failures that may be manifestations of the same fault, Apollo records the output statement (i.e., `echo` or `print`) that generated each fragment of HTML output.

State Manager. PHP applications make use of persistent state such as the database, session information, and cookies. The State Manager is in charge of (i) restoring the environment prior to each execution, and (ii) storing the new environment after each execution.

3.4.2 Bug Finder

The bug finder is in charge of transforming the results of the executed inputs into bug reports. Below is a detailed description of the components of the bug finder.

Bug Report Repository This repository stores the bug reports found in all executions. Each time a failure is detected, the corresponding bug report (for all failures with the same characteristics) is updated with the path constraint and the input inducing the failure. A failure is defined by its characteristics, which include: the type of the failure (execution failure or HTML failure), the corresponding message (PHP error/warning message for execution failures, and validator message for HTML failures), and the PHP statement generating the problematic HTML fragments identified by the validator (for HTML failures), or the PHP statement involved in the PHP interpreter

[10]On the other hand, any data flow that passes outside PHP, such as via JavaScript code in the generated HTML, will not be tracked by this approach.

error report (for execution failures). When the exploration is complete, each bug report contains one failure characteristic, (error message and statement involved in the failure) and the sets of path constraints and inputs exposing failures with the same characteristics.

Oracle. PHP web applications output HTML/XHTML. Therefore, in Apollo, we use as oracle an HTML validator that returns syntactic (malformed) HTML failures found in a given document. We experimented with both the offline WDG validator[11] and the online W3C markup validation service[12]. Both oracles identified the same HTML failures. Our experiments use the faster WDG validator.

Input Minimizer. Apollo implements the algorithm described in Figure 3-3 to perform *post-mortem* minimization of the path constraints. For each bug report, the minimizer executes the program multiple times, with multiple inputs that satisfy different path constraints, and attempts to find the shortest path constraint that results (executing the program with an input satisfying the path constraint) in the same failure characteristics.

3.4.3 Input Generator

UI Option Analyzer

Many PHP web applications create interactive HTML pages that contain user interface elements such as buttons and menus that require user interaction to execute further parts of the application. In such cases, pressing the button may result in the execution of additional PHP source files. There are two challenges involved in dealing with such interactive applications. First, we need to analyze the HTML output to find the referenced scripts, and the different values that can be supplied as parameters. Second, Apollo needs to be able to follow input parameters through the shared global information (database, the `session`, and the `cookie` mechanisms)

Apollo's approach to the above challenges is to simulate user interaction by analyzing the dynamically created HTML output, and tracking the symbolic parameters through the environment (with the exception of the database). Apollo automatically extracts the available user options from the HTML output. Each option contains the script to execute, along with any parameters (with default values if supplied) for that script. Apollo also analyzes recursive static HTML documents that can be called from the dynamic HTML output, i.e. Apollo traverses hyperlinks in the generated dynamic HTML that link to other HTML documents on the same site.

Since additional code on the client side (for instance, Java script) might be executed when a button is pressed, this approach might induce false positive bug reports. In our experiments, this limitation produced no false positive bug reports.

For example after analyzing the output of the program of Figure 3-8, the UI Option Analyzer will return the following two options:

1. Script: "mainmenu.php"
   PathConstraint: `txtNick` = "*Admin*" $\wedge$ Exist (`pwdPassword`)

[11] `http://htmlhelp.com/tools/validator/offline`
[12] `http://validator.w3.org`

```
<?php
   echo "<h2>WebChess ".$Version." Login"</h2>;
?>
<form method="post" action="mainmenu.php">
<p>
   Nick: <input name="txtNick" type="text" size="15" default="admin"/>
   <br />
   Password: <input name="pwdPassword" type="password" size="15"/>
</p>
<p>
   <input name="login" value="login" type="submit"/>
   <input name="newAccount" value="New Account"
     type="button" onClick="window.open('newuser.php', '_self')"/>
</p>
</form>
```

Figure 3-8: A simplified version of the main entry point (`index.php`) to a PHP program. The HTML output of this program contains a form with two buttons. Pressing the `login` button executes `mainmenu.php` and pressing the `newAccount` button will execute the `newuser.php` script.

2. Script: “newuser.php”
   PathConstraint: ∅

The **Symbolic Driver** implements the combined concrete and symbolic algorithm of Figure 3-2. The driver has two main tasks: select which input to consider next (line 5), and create additional inputs from each executed input (by negating conjuncts in the path constraint). To select which input to consider next, the driver uses a *coverage heuristic*, similar to those used in EXE [27] and SAGE [68]. Each conjunct in the path constraint knows the branch that created the conjunct, and the driver keeps track of all branches previously executed and favors inputs created from path constraints that contain un-executed branches.

To avoid redundant exploration of similar executions, Apollo performs state matching (performed implicitly in Line 11 of Figure 3-6) by not adding already-explored transitions.

Constraint Solver. The interpreter implements a lightweight symbolic execution, in which the only constraints are equality and inequality with constants. Apollo transforms path constraints into integer constraints in a straightforward way, and uses `choco`[13] to solve them.

This approach still allows us to handle values of the standard types (integer, string), and is straightforward because the only constraints are equality and inequality[14].

In cases where parameters are unconstrained, Apollo uses a combination of values that are randomly generated and values that are obtained by mining the program text for constants (in particular, constants used in comparison expressions).

[13] `http://choco-solver.net/index.php?title=Main_Page`

[14] Floating-point values can be handled in the same way, though none of the examined programs required it.

program	version	#files	total LOC	PHP LOC	#downloads
faqforge	1.3.2	19	1,712	734	14,164
webchess	0.9.0	24	4,718	2,226	32,352
schoolmate	1.5.4	63	8,181	4,263	4,466
phpsysinfo	2.5.3	73	16,634	7,745	492,217
timeclock	1.0.3	62	20,792	13,879	23,708
phpBB2	2.0.21	78	34,987	16,993	18,668,112

Figure 3-9: Characteristics of subject programs. The **#files** column lists the number of .php and `.inc` files in the program. The **total LOC** column lists the total number of lines in the program files. The **PHP LOC** column lists the number of lines that contain executable PHP code. The **#downloads** column lists the number of downloads from `http : //sourceforge.net`.

3.5 Evaluation

We experimentally measured the effectiveness of Apollo by using it to find faults in PHP web applications. We designed experiments to answer the following research questions:

Q1. How many faults can Apollo find, and of what varieties?

Q2. How effective is the fault detection technique of Apollo compared to alternative approaches in terms of the number and severity of discovered faults and the line coverage achieved?

Q3. How effective is our minimization technique in reducing the size of input parameter constraints and failure-inducing inputs?

For the evaluation, we selected 6 open-source PHP programs from `http://sourceforge.net` (see Figure 3-9):

- **faqforge**: tool for creating and managing documents.
- **webchess**: online chess game.
- **schoolmate**: PHP/MySQL solution for administering elementary, middle, and high schools.
- **phpsysinfo**: displays system information, e.g., uptime, CPU, memory, etc.
- **timeclock** is a web-based timeclock system.
- **phpBB2** is a discussion forum.

3.5.1 Generation Strategies

We use the following test input generation strategies in the remainder of this section:

- **Apollo** generates test inputs using the technique described in Section 3.2.

program	strategy	#inputs generated	% line coverage	execution crash	execution error	execution warning	HTML error	HTML warning	Total faults
faqforge	Rand	1461	19.2	0	0	0	10	1	11
	Apollo	717	92.4	0	9	0	46	19	74
webchess	Rand	1805	5.9	1	13	2	3	0	19
	Apollo	557	42.0	1	20	2	7	0	35
schoolmate	Rand	1396	8.3	1	0	0	18	0	19
	Apollo	724	64.9	2	21	9	58	0	100
phpsysinfo	Rand	406	21.3	0	5	3	2	0	10
	Apollo	143	56.2	0	5	4	2	0	11
timeclock	Rand	928	3.2	0	1	1	29	1	32
	Apollo	789	14.1	0	1	1	64	1	67
phpbb2	Rand	2497	11.4	0	0	3	1	0	4
	Apollo	649	31.7	0	0	5	21	0	26
Total	Rand	8493	11.6	2	19	9	63	2	95
	Apollo	3579	50.2	3	56	21	198	20	302

Figure 3-10: Experimental results for 10-minute test generation runs. The table presents results for each subject program, and each strategy, separately. The **#inputs** column presents the number of inputs that each strategy created in the given time budget. The **coverage** column lists the line coverage achieved by the generated inputs. The **execution crashes**, **errors**, **warnings** and **HTML errors**, **warnings** columns list the number of faults in the respective categories. The **Total faults** columns sums up the number of discovered faults.

- **Randomized** is an approach similar to the one proposed by Halfond and Orso [72] for JavaScript. The test input generation strategy generates test inputs by giving random values to parameters. The values are chosen from constant values that appear textually in the program source and from default values. A difficulty is that the parameters' names and types are not immediately clear from the source code. The randomized strategy infers the parameters' names and types from dynamic traces—any variable for which the user can supply a value, is classified as a parameter.

3.5.2 Methodology

To answer the first research question (Q1) we applied Apollo to 6 subject programs and we classified the discovered failures into five groups based on their different failure characteristics:

- **execution crash:** the PHP interpreter terminates with an exception.
- **execution error:** the PHP interpreter emits an error message that is visible in the generated HTML.
- **execution warning:** the PHP interpreter emits an error message that is invisible in the generated HTML.
- **HTML error:** the program generates HTML for which the validator produces an error report.

- **HTML warning:** the program generates HTML for which the validator produces a warning report.

This classification is a refinement of the one presented in Section 3.1.3.

To answer the second research question (Q2) we compared our technique to two other approaches. We compared both the coverage achieved and the number of faults found with the **Randomized** generation strategy. Coverage was measured using the line coverage metric, i.e., the ratio of the number of executed lines to the total number of lines with executable PHP code in each application. We ran each test input generation strategy for 10 minutes on each subject program. This time limit was chosen arbitrarily, but it allows each strategy to generate hundreds of inputs and we have no reason to believe that the results would be materially affected by a different time limit. This time budget includes all experimental tasks, i.e., program execution, harvesting of constant values from program source, test generation, constraint solving (where applicable), output validation via an oracle, and line coverage measurement. To avoid bias, we ran both strategies inside the same experimental harness. This includes the Database Manager (Section 3.4), which supplies user names and passwords for database access. For our experiments, we use the WDG offline HTML validator, version 1.2.2.

We also compared Apollo's results to the results reported by Minamide's static analysis [99] on four subject programs (Section 3.5.3.1 presents the results).

To answer the third research question, about the effectiveness of the input minimization, we performed the following experiments. Recall that several execution paths and inputs may expose the same failure. Our input minimization algorithm attempts to produce the shortest possible input that exposes each failure. The inputs to the minimizer are the failure found by the algorithm in Figure 3-6 along with all the execution paths that expose each failure.

3.5.3 Results

Figure 3-10 tabulates the faults (we manually inspected most of the reported failures and, to the best of our knowledge, all reported faults are counted only once). and line coverage results of running the two test input generation strategies on the subject programs. The **Apollo** strategy found 302 faults in the subject applications, versus only 95 faults for **Randomized**. Moreover, the **Apollo** test generation strategy achieved an average line coverage of 50.2%, versus only 11.6% for **Randomized**.

The coverage of **phpbb2** and **timeclock** is relatively small as the output of these applications contains client-side scripts written in JavaScript which Apollo currently does not analyze.

Figures 3-11 and 3-12 classify the faults reported by Apollo. The execution errors (Figure 3-11) are dominated by database-related errors, where the application had difficulties accessing the database, resulting in error messages such as (1) "supplied argument is not a valid MySQL result resource" and (2) "Unable to jump to row 0 on MySQL result". The two SQL-related error messages quoted above occurred in faqforge (9 cases of error 1) and webchess (19 cases of error 1 and 1 case of error 2), schoolmate (20 cases of error 1 and 9 cases of error 2), timeclock (1 case of error 1), and phpbb2 (1 case of error 1).

Fault Category	Faults	Percentage
Malformed SQL	60	71.4
Array index out of bound	5	6.0
Resource used as offset	4	4.8
Failed to open stream	4	4.8
File not found	2	2.6
Can't open connection	2	2.6
Assigning reference	2	2.6
Undefined function	1	1.2

Figure 3-11: Classification of the execution faults found by Apollo.

Fault Category	Faults	Percentage
Element not allowed	40	17.5
Missing end tag	39	17.1
Can't generate system identifier	25	11.0
No attribute	25	11.0
Unopened close tag	21	9.2
Missing attribute	21	9.2
character not allowed	11	4.8
End tag for unfinished element	11	4.8
Incorrect attribute	8	3.5
Element declaration finished prematurely	8	3.5
Unfinished tag	7	3.1
Duplicate specification	4	1.8
Undefined element	4	1.8
Incorrect attribute value	4	1.8

Figure 3-12: Classification of the HTML faults found by Apollo.

These failures have the same cause: user-supplied input parameters are concatenated directly into SQL query strings; leaving these parameters blank results in malformed SQL causing the `mysql_query` functions to return an invalid result. The subject programs failed to check the return value of `mysql_query`, and simply assume that a valid result is returned. These faults are indications of a potentially serious problem: the concatenation of user-supplied strings into SQL queries makes these programs vulnerable to SQL injection attacks [45]. Thus our testing approach indicates possible SQL injection vulnerabilities despite not being specifically designed to look for security issues.

The three execution crashes (when the interpreter terminates with an exception) in Figure 3-10 happen when the interpreter tries to load files or functions that are missing. For instance, for some inputs that can be supplied to the schoolmate subject program, the PHP interpreter attempts to load a file that does not exist in the current distribution of schoolmate. Since schoolmate has 63 files,

and PHP is an interpreted language that allows the use of run-time string values when loading files, it is hard to detect such faults. Apollo also discovers a severe fault in the webchess subject program. This fault occurs when the interpreter tries to call to a function that is undefined since the PHP file implementing it is not included due to a value supplied as one of the parameters.

The 228 malformed HTML faults can be divided into several categories (Figure 3-12), These faults are mainly concerned with HTML elements that occur in the wrong place, HTML elements with incorrect values, and with unclosed tags. The breakdown of HTML faults is similar across the different PHP applications.

3.5.3.1 Comparison with Static Analysis

Minamide [99] presents a static analysis for discovering HTML malformedness faults in PHP applications. Minamide's analysis tool approximates the string output of a program with a context-free grammar, then discovers unclosed tags by intersecting this grammar with the regular expression of matched pairs of delimiters (open/closed tags). By contrast, our analysis uses an HTML validator and covers the entire language standard.

We performed our evaluation on a set of applications overlapping with Minamide's (webchess, faqforge, schoolmate, timeclock). For these four overlapping subject programs, Apollo is both more *effective* and more *efficient* than Minamide's tool. Apollo found 3.4 times as many HTML validation faults as Minamide's tool (195 vs. 56). The faults found by Minamide's tool are not publicly available so we do not know whether Apollo discovered all faults that Minamide's tool discovered. However, Apollo found 80 execution faults, which are out of reach for Minamide's tool. Apollo is also more scalable—on schoolmate, Apollo found 58 malformed HTML faults in 10 minutes, while Minamide's tool found only 14 faults in 126 minutes. The considerably longer running time of Minamide's tool is due to the construction of large automata and to the expensive algorithm for checking disjointness between regular expressions and context-free languages.

3.5.3.2 Path Constraint Minimization

We measure the effectiveness of the minimization algorithm of Section 3.2.3 via the reduction ratio between the size of the shortest original (un-minimized) path constraint (and input) and the minimized path constraint (and input).

Figure 3-13 tabulates the results. The results show that our input minimization technique effectively reduces the size of inputs by at least 42%, for more than 50% of the faults.

3.5.4 Threats to Validity

Construct Validity. Why do we count malformed HTML as a defect in dynamically generated webpages? Does a webpage with malformed HTML pose a real problem or this is an artificial problem generated by the overly conservative specification of the HTML language? Although web browsers are resilient to malformed HTML, we have encountered cases when malformed HTML crashed the popular Internet Explorer web browser. More importantly, even though a particular browser might tolerate malformed HTML, different browsers or different versions of the

program	success rate%	path constraints orig. size	path constraints reduction	inputs orig. size	inputs reduction
faqforge	64	22.3	78%	9.3	69%
webchess	91	23.4	81%	10.9	60%
schoolmate	51	22.9	62%	11.5	42%
phpsysinfo	82	24.3	82%	17.5	74%

Figure 3-13: Results of minimization. The **success rate** indicates the percentage of faults whose exposing input was successfully minimized (i.e., the minimizer found a shorter exposing input). The **orig. size** columns list the average size of original (un-minimized) path constraints and inputs. The size of a path constraint is the number of conjuncts. The size of an input is the number of key-value pairs in the input. The **reduction** columns list the amount by which the minimized size is smaller than the unminimized size (i.e., $1 - \frac{minimized}{unminimized}$). The higher the percentage, the more successful the minimization.

same browser may not display all information in the presence of malformed HTML. This becomes crucial for some websites, for example for sites related to financial transactions. Many websites provide a button for verifying the validity of statically generated HTML. The challenges of dynamically generated webpages prevent the same institutions from validating the content.

Why do we use line coverage as a quality metric? We use line coverage only as a *secondary* metric, our *primary* metric being the number of faults found. Line coverage indicates how much of the application was explored by the analysis. An analysis can only find faults in lines that are covered, so more coverage generally leads to more faults being detected.

Why do we present the user with minimized path constraints and inputs in addition to the inputs exposing the failure? Although an input that corresponds to a longer path constraint still exposes the same failure, in our experience, the removal of superfluous information helps programmers with pinpointing the location of the fault.

Internal Validity. Did Apollo discover real, unseeded, and unknown faults? Since we used subject projects developed by others, we could not influence the quality of the subject programs. Apollo does not search for known or seeded faults, but it finds *real* faults in real programs. For those subject programs that connect to a database, we populated the database with random records. The only thing that is "seeded" into the experiment is a username/password combination, so that Apollo can access the records stored in the database.

External Validity. Will our results generalize beyond the subject programs? We only used Apollo to find faults in 6 PHP projects. These may have serious quality problems, or may be unrepresentative in other ways. Four of the subject programs were also used as subject programs by Minamide [99]. We chose the same programs to compare our results. We chose an additional subject program, phpsysinfo, since it is almost double the size of the largest subject that Minamide used. Additionally, phpsysinfo is a mature and active project in sourceforge. It is widely used, as witnessed by almost half a million downloads (Figure 3-9), and it is ranked in the top 0.5% projects on sourceforge (rank 997 of 176,575 projects as of 7 May 2008). Nevertheless, Apollo found 11 faults in phpsysinfo.

Reliability. Are the results reproducible? The subject programs that we used are publicly available from sourceforge. The faults that we found are available for examination at `http://pag.`

csail.mit.edu/apollo.

3.5.5 Limitations

Simulating user inputs based only locally executed JavaScripts. In general, the HTML output of a PHP script might contain buttons and arbitrary snippets of JavaScript code that are executed when the user presses the corresponding button. The actions that the JavaScript might perform are currently not analyzed by Apollo. For instance, the JavaScript code might pass specific arguments to the PHP script. As a result, Apollo might report false positives. Apollo might report a false positive if Apollo decides to execute a PHP script as a result of simulating a user pressing a button that is not visible. Apollo might also report a false positive if it attempts to set an input parameter that would have been set by the JavaScript code. In our experiments, Apollo did not report any false positives.

Limited tracking in native methods. Apollo has limited tracking of input parameters through PHP native methods. PHP native methods are implemented in C, which make it difficult to automatically track how input parameters are transformed into output parameters. We have modified the PHP interpreter to track parameters across a very small subset of the PHP native methods. Similar to [151], we plan to create an external language to model the dependences between inputs and outputs for native methods to increase Apollo line coverage when native methods are executed.

3.6 Related Work

In this section, we discuss three categories of related work: (i) combined concrete and symbolic execution, (ii) techniques for input minimization, and (iii) testing of web applications.

3.6.1 Combined Concrete and Symbolic Execution

DART [67] is a tool for finding combinations of input values and environment settings for C programs that trigger errors such as assertion failures, crashes and nontermination. DART combines random test generation with symbolic reasoning to keep track of constraints for executed control-flow paths. A constraint solver directs subsequent executions towards uncovered branches. Experimental results indicate that DART is highly effective at finding large numbers of faults in several C applications and frameworks, including important and previously unknown security vulnerabilities. CUTE [133] is a variation (called *concolic testing*) on the DART approach. The authors of CUTE introduce a notion of approximate pointer constraints to enable reasoning over memory graphs and handle programs that use pointer arithmetic.

Subsequent work extends the original approach of combining concrete and symbolic executions to accomplish two primary goals: 1) improving scalability [8,25,65,66,68,94], and 2) improving execution coverage and fault detection capability through better support for pointers and arrays [27, 133], better search heuristics [68,81,93], or by encompassing wider domains such as database applications [57].

Godefroid [65] proposed a compositional approach to improve the scalability of DART. In this approach, summaries of lower level functions are computed dynamically when these functions are first encountered. The summaries are expressed as pre- and post-conditions of the function in terms of its inputs. Subsequent invocations of these lower level functions reuse the summary. Anand [8] extends this compositional approach to be demand-driven to reduce the summary computation effort.

Exploiting the structure of the program input may improve scalability [66, 94]. Majumdar and Xu [94] abstract context-free grammars that represent the program inputs to produce a symbolic grammar. This grammar reduces the number of input strings to enumerate during test generation.

Majumdar and Sen [93] describe hybrid concolic testing, interleaves random testing with bounded exhaustive symbolic exploration to achieve better coverage. Inkumsah and Xie [81] combine evolutionary testing using genetic mutations with concolic testing to produce longer sequences of test inputs. SAGE [68] also uses improved heuristics, called *white-box fuzzing*, to achieve higher branch coverage.

Emmi [57] extends concolic testing to database applications. This approach creates and inserts database records and enables testing program code that depends on embedded SQL queries.

Wassermann [152] presents a concolic testing tool for PHP. The goal of his work is to automatically identify security vulnerabilities caused by injecting malicious strings into SQL commands. Their tool uses a framework of finite-state transducers and a specialized constraint solver.

Some approaches aim at checking functional correctness. A number of tools [23, 26] use a separate implementation of the function being tested to compare outputs. This limits the approach to situations where a second implementation exists.

While our work builds on this significant body of research, there are two significant differences. First, our work goes beyond simple assertion failures and crashes by using on an oracle (in the form of an HTML validator) to determine correctness, which means that our tool can handle situations where the program has functionally incorrect behavior without relying on programmer assertions. Second, our work addresses PHP's complex execution model, that involves multiple scripts invoked via user-interface options in generated HTML pages, and communicating values via session state and cookies. The only other concolic testing approach for PHP [152] does not present a fully automatic solution for dealing with multiple interrelated PHP scripts.

3.6.2 Minimizing Failure-Inducing Inputs

Our work minimizes the constraints on the input parameters. This shortens the failure-inducing inputs and to help to pinpoint the cause of faults. Godefroid et al. [68] faced this challenge since their technique produces several distinct inputs that expose the same fault. Their approach hashes all such inputs and returns an example failure-inducing input. Our work also addresses another issue: identifying the minimal set of program variables that are essential to induce the failure. In this regard, our work is similar to *delta debugging* [34, 162] and its extension *hierarchical delta debugging* [100]. These approaches modify the failure inducing input directly, thus leading to a single, minimal failure-inducing input. In contrast, our technique modifies the set of constraints on the failure-inducing input. This creates minimal *patterns* of failure-inducing inputs, which

facilitates debugging. Moreover, our technique is more efficient, because it takes advantage of the (partial) overlapping of different inputs.

3.6.3 Testing of Web Applications

The language under consideration in this chapter, PHP, is quite different from the focus of previous testing research. PHP poses several new challenges such as dynamic inclusion of files, and function definitions that are statements. Existing techniques for fault detection in PHP applications use static analysis and target security vulnerabilities such as *SQL injection* or *cross-site scripting* (XSS) attacks [79, 84, 99, 150, 158]. In particular, Minamide [99] uses static string analysis and language transducers to model PHP string operations to generate *potential* HTML output—represented by a context-free grammar—from the web application. This method can be used to generate HTML document instances of the resulting grammar and to validate them using an existing HTML validator. As a more complete alternative, Minamide proposes a *matching validation* which checks for containment of the generated context free grammar against a regular subset of the HTML specification. However, this approach can only check for matching start and end tags in the HTML output, while our technique covers the entire HTML specification. Also, flow-insensitive and context-insensitive approximations in the static analysis techniques used in this method result in false positives, while our method reports only real faults.

Kieżun presents a dynamic tool, Ardilla [85], to create SQL and XSS attacks. Their tool uses dynamic tainting, concolic execution, and attack-candidate generation and validation. Like ours, his tool reports only real faults. However, Kieżun focuses on finding security faults, while we concentrate on functional correctness. His tool builds on and extends the input-generation component of Apollo but does not address the problem of user interaction. It is an interesting area of future research to combine Apollo's user-interaction and state-matching with Ardilla's exploit-detection capabilities.

McAllister et al. [96] also tackle the problem of testing interactive web application. Their approach attempts to follow user interactions. Their method relies on pre-recorded traces of user interactions, while our approach automatically discovers allowable interactions. Moreover, their approach to handling persistent state relies on instrumenting one particular web application framework, Django. In contrast, our approach is to instrument the PHP run-time system and observe database interactions. This allows handling state of PHP applications regardless of any framework they may use.

Benedikt et al. [17] present a tool, VeriWeb, for automatically testing dynamic webpages. They use a model checker to systematically explore all paths (up to a certain bound) of user navigate in a web site. When the exploration encounters HTML forms, VeriWeb uses *SmartProfiles*. SmartProfiles are user-specified attribute-value pairs that are used to automatically populate forms and supply values that should be provided as inputs. Although VeriWeb can automatically fill in the forms, the human tester needs to pre-populate the user profiles with values that a user would provide. In contrast, Apollo automatically discovers input values by looking at the branch conditions along an execution path. Benedikt et al. do not report any faults found, while we report 302.

Dynamic analysis of string values generated by PHP web applications has been considered in a *reactive* mode to prevent the execution of insidious commands (*intrusion prevention*) and to raise

an alert (*intrusion detection*) [82, 115, 139]. As far as we know, our work is the first attempt at *proactive* fault detection in PHP web applications using dynamic analysis.

Finally, our work is closely related to *implementation based* (as opposed to *specification based* e.g., [123]) testing of web applications. These works abstract the application behavior using a) client-side information such as user requests and corresponding application responses [54, 60], or b) server-side monitoring information such as user session data [56, 135], or c) static analysis of server-side implementation logic [72]. The approaches that use client-side information or server-side monitoring information are inherently incomplete, and the quality of generated abstractions depends on the quality of the tests run.

Halfond and Orso [72] use static analysis of the server-side implementation logic to extract a web application's interface, i.e., the set of input parameters and their potential values. They implemented their technique for JavaScript. They obtained better code coverage with test cases based on the interface extracted using their technique as compared to the test cases based on the interface extracted using a conventional web crawler. However, the coverage may depend on the choices made by the test generator to combine parameter values—an exhaustive combination of values may be needed to maximize code coverage. In contrast, our work uses dynamic analysis of server side implementation logic for fault detection and minimizes the number of inputs needed to maximize the coverage. Furthermore, we include results on fault detection capabilities of our technique. We implemented and evaluated (Section 3.5) a version of Halfond and Orso's technique for PHP. Compared to that re-implementation, Apollo achieved higher line coverage (50.2% vs. 11.6%) and found more faults (302 vs. 95).

3.7 Conclusions

We have presented a test generation technique that helps testers detect faults in PHP web applications. Our technique is based on combined concrete and symbolic execution. The work is novel in several respects. First, the technique not only detects run-time errors but also uses an HTML validator as an oracle to determine situations where malformed HTML is created. Second, we address a number of PHP-specific issues, such as the simulation of interactive user input that occurs when user interface elements on generated HTML pages are activated, resulting in the execution of additional PHP scripts. Third, we perform an automated analysis to minimize the size of failure-inducing inputs.

We created a tool, Apollo, that implements the analysis. We evaluated Apollo on 6 open-source PHP web applications. Apollo's test generation strategy achieves over 50% line coverage. Apollo found a total of 302 faults in these applications: 84 execution problems and 218 cases of malformed HTML. Finally, Apollo also minimizes the size of failure-inducing inputs: the minimized inputs are up to 5.3× smaller than the unminimized ones.

Chapter 4

Elimination

This chapter presents a test generation technique to help maintainers eliminate bugs by automatically reproducing failures. It is almost impossible to find and eliminate a software failure, and to verify the solution, without the ability to reproduce the failure. However, reproducing a failure is often difficult and time-consuming.

As an example, consider bug #30280 from the Eclipse bug database (Figure 4-1). A user found a crash and supplied a back-trace, but neither the developer nor the user could reproduce the problem. Two days after the bug report, the developer finally reproduced the problem; four minutes after reproducing the problem, the developer fixed it.

Our work aims to reduce the amount of time it takes a developer to reproduce a problem. Our technique, ReCrash, generates multiple unit tests that reproduce an observed failure. For example, suppose that the user (in Figure 4-1) had been using a ReCrash-enabled version of Eclipse. As soon as the Eclipse crash occurred, ReCrash would have generated a set of unit tests (Figure 4-2), each of which reproduces the problem. The user could have sent these test cases with the initial bug report, eliminating the two days delay to reproduce the problem.

Upon receiving the test cases, the developer could run them under a debugger to examine fields, step through execution, or otherwise investigate the cause of failure. (The readability of the test case is secondary to reproducibility; a test need not be readable, nor end-to-end, to be useful.) The developer can use the same test to later verify his proposed solution for the bug.

Reproducing failures can be difficult for the following reasons.

Nondeterminism A problem that depends on timing (e.g., context switching), memory layout (e.g., hash codes), or random number generators will manifest itself only rarely. Reproducing the problem requires replacing the sources of nondeterminism with specific choices that reproduce previous behavior. As an example, consider the (somewhat contrived) RandomCrash program [143] of Figure 4-5.

Remote detection A problem that is discovered by someone other than the developer who can fix it may depend on local information such as user GUI actions, environment variables, the state of the file system, operating system behavior, and other explicit or implicit program inputs. Not all dependences may be apparent to the developer or the user; many users are not

2003-01-27 08:01	U:	I found a crash (back-trace supplied)
2003-01-27 08:26	D:	Which build are you using? Do you have a test-case to reproduce?
2003-01-27 08:39	D:	Which JDK are you using?
2003-01-28 13:06	U:	I'm running eclipse 2.1, ... I was not able to reproduce the crash
2003-01-29 04:28	D:	Thanks for clarification ...
2003-01-29 04:33	D:	Reproduced.
2003-01-29 04:37	D:	Fixed.

Figure 4-1: An excerpt of comments (`https://bugs.eclipse.org/bugs/show_bug.cgi?id=30280`) between the user (U) who reported the Eclipse bug #30280 and the developer (D) who fixed it.

sophisticated enough to gather this information; some of the information may be confidential; and the effort of collecting it may be too burdensome for the user, the developer (during interactions with the user), or both.

Test case complexity Even if a problem can be reproduced deterministically, the exposing execution might be complex, and the buggy method might be called multiple times before the bug is triggered. A simpler test case, such as a unit test, is often faster and easier to run and understand than the execution that triggered the failure.

We propose a technique, ReCrash, that help maintainers eliminate bugs, by generating tests that reproduce executions in many of the above cases. ReCrash automatically converts a failing program execution into a set of deterministic and self-contained unit tests. Each of the unit tests reproduces the problem that caused the program to fail.

ReCrash has two phases: monitoring and test case generation.

monitoring During an execution of the target program, ReCrash maintains a shadow stack with an n-deep copy of the receiver and arguments to each method. From depth n on, these objects refer to the original objects on the heap (see Section 4.1.2.1). Thus, ReCrash exploits the object-oriented nature of the program by using objects as the unit of granularity for including or excluding values from the stored shadow stack. When the program fails (i.e., crashes), ReCrash serializes the shadow stack contents, including all heap objects referred to from the shadow stack. Heap data that was not copied to the shadow stack might have been side-effected between the point of the call and the point of the failure; this is a cause of imprecision for the ReCrash technique.

test generation ReCrash generates candidate tests by calling each method that was on the shadow call stack with the de-serialized receiver and arguments from the shadow stack. ReCrash outputs all candidate tests that reproduce the original failure. ReCrash outputs multiple tests because tests that call only the methods that were at top of the call stack may not provide enough context to find the error, while tests calling methods that are closer to the bottom of the call-stack provide more context, but are less likely to reproduce the original failure.

```
1  // Generated by reCrash
2  // Eclipse 2.1M4/JDK 1.4
3  //-- The original crash stack back trace for java.lang.NullPointerException:
4  //   org...QualifiedAllocationExpression.resolveType
5  //   org...Expression.resolve
6  //   ...
7  public class EclipseTest extends TestCase {
8    protected void setUp() throws Exception {
9      ShadowStackReader.readTrace("eclipse.trace");
10   }
11
12   public void test_resolveType() throws Throwable {
13     ShadowStackReader.setMethodStackItem(0);
14
15     // load receiver
16     // rec = i.new Y()
17     QualifiedAllocationExpression rec =
18       (QualifiedAllocationExpression)ShadowStackReader.readReceiver(0);
19
20     // load arguments
21     // arg_1 = Scope-locals: java.lang.String;
22     BlockScope arg_1 = (BlockScope)ShadowStackReader.readArgument(1);
23
24     // Method invocation
25     rec.resolveType(arg_1);
26  }
27
28   public void test_resolve() throws Throwable {
29     ShadowStackReader.setMethodStackItem(1);
30
31     // load receiver
32     // rec = i.new Y()
33     Expression rec = (Expression)ShadowStackReader.readReceiver(0);
34
35     // load arguments
36     // arg_1 = Scope-locals: java.lang.String;
37     BlockScope arg_1 = (BlockScope)ShadowStackReader.readArgument(1);
38
39     // Method invocation
40     rec.resolve(arg_1);
41   }
42 }
```

Figure 4-2: Two test cases (out of eight) generated by ReCrash to reproduce Eclipse bug #30280. Each test case loads the method arguments from a serialized representation stored in `eclipse.trace`(line 9). Lines 16, 21, 32, 36 show the `toString` representation of the object being read from the serialized trace.

ReCrash is effective, despite recording only partial information about the program state in-memory. For many failures, it is possible to reproduce them with *only* some of the information available on entry to the methods on the stack at the time of the failure. This may be due to the following characteristics of object-oriented programs.

- Many bugs are dependent on small parts of the heap (also the premise of unit testing).
- Good object-oriented style encapsulates important state nearby.
- Good object-oriented style avoids excessive use of globals. Furthermore, ReCrash has access to and will store any parts of the global state or environment that are passed as method arguments.

ReCrash's monitoring phase can be made efficient by reducing the amount of monitoring. For example, objects that are not changed by the code need not be monitored. As another example, ReCrash has an optimization mode called *second chance* in which the monitoring phase initially records only stack back-traces when the program fails. On subsequent runs, ReCrash monitors only the methods that were in the stack back-trace of failures. If the same problem reappears, it will be captured and reproduced. Second chance allows a vendor to use ReCrash in a similar way to an anti-virus program that frequently updates the virus profile data. When one client discovers a failure and sends the produced back-trace to the vendor, the vendor will send an update containing a list of additional methods to monitor, to all other clients. When ReCrash is used remotely (in the field, not by the developer who is debugging a failure), the test generation phase may be performed either remotely or by the developer.

This chapter makes the following contributions:

- The ReCrash technique efficiently captures and reproduces failures using in-memory storage of method arguments.
- Optimizations give the ReCrash technique low enough overhead to enable routine use, by monitoring only relevant parts of a program.
- The second chance mode operates with almost no overhead.
- ReCrashJ is a practical implementation of ReCrash for Java.
- Case studies show that ReCrashJ effectively and efficiently reproduces failures for several real applications.

The remainder of this chapter is organized as follows. Section 4.1 describes the ReCrash technique. Section 4.2 presents the ReCrashJ implementation. Section 4.3 describes our experimental evaluation. Section 4.4 discusses some of ReCrash's limitations. Section 4.5 surveys related work, and Section 4.6 concludes.

4.1 ReCrash Technique

The ReCrash technique has two parts. Monitoring is done during program execution (Section 4.1.1), and test generation is done after the program fails (Section 4.1.3). Section 4.1.2 discusses several optimizations to the monitoring phase.

4.1.1 Monitoring Phase

In the monitoring phase, ReCrash maintains in memory a shadow version of the call stack with copies of the receiver and arguments to each method. Figure 4-3 presents pseudo-code for the monitoring phase.

On entry to method m with arguments $args$, ReCrash generates a unique identifier id for the invocation of m, then pushes $\langle id, \mathtt{m}, rec_{copy}, args_{copy} \rangle$ onto the ReCrash shadow stack, where rec_{copy} contains copies of the method's receiver and $args_{copy}$ contains a copy of the method's arguments.

ReCrash can use different copy strategies. It can make a deep copy of each argument, store only a reference, or use hybrid strategies (see Section 4.1.2.1). If m exits normally, ReCrash removes the method call data $\langle id, \mathtt{m}, rec_{copy}, args_{copy} \rangle$ from the top of its shadow stack.

If an un-handled exception is thrown out of `main`, ReCrash outputs the un-handled exception and a deep copy of the current ReCrash shadow stack. The methods that ReCrash stores include at least the methods on the virtual machine call stack at the time of the failure. In cases where a method catches an exception and then throws a different uncaught exception, the ReCrash shadow stack contains additional methods (those that were on the shadow stack when the original exception was thrown). This additional information can improve the reproducibility of failures, as ReCrash will be able to reproduce system states before the time the first exception was thrown.

4.1.2 Optimizations to the Monitoring Phase

The cost of copying the method arguments at method entry dominates ReCrash's time and space overhead. We have considered two orthogonal ways to reduce overhead by recording less information: reducing the depth of copied state for the method's arguments, and monitoring fewer methods. Recording less information might reduce the chances of reproducing a failure. Section 4.3 discusses tradeoffs between the performance overhead and the ability to reproduce failures.

4.1.2.1 Depth of Copying Arguments

In order to be able to recreate the state of the method's arguments if the method fails, ReCrash copies parts of the arguments to the shadow stack upon method entry. However, a deep copy of the shadow stack is only available after an exception is not caught by the program. Thus, any part of an argument's state that is not copied to the shadow stack may change between the method entry and the time of the failure. This change might prevent ReCrash from reproducing the failure.

This section considers different strategies for copying arguments (including the receiver) at the method entry. In each strategy a different amount of the argument state is copied to the shadow

Input: `copy` - a function that copies arguments (see Section 4.1.2.1)
Output: `file` - containing deep copy of the shadow stack

s : shadow stack of the current method on the call stack and their arguments

On program start:
begin
  $s \leftarrow$ new empty stack
end

On entry to method m in the call `rec.m(args)`:
begin
  $rec_{copy} \leftarrow$ `copy`(rec)
  $args_{copy} \leftarrow$ `copy`(args)
  $id \leftarrow$ generate unique id for current execution of m
  push tuple $\langle id, \mathtt{m}, rec_{copy}, args_{copy} \rangle$ into s
end

On non-exceptional exit from method m:
begin
  $id \leftarrow$ lookup id for current execution of m
  pop from s until $\langle id, _, _, _ \rangle$ is popped
end

On top-level (main) uncaught exception e:
begin
  $file \leftarrow$ new output file
  store e to $file$
  foreach $\langle id, \mathtt{m}, rec_{copy}, args_{copy} \rangle$ in s **do**
    store $\langle id$, m, `deepCopy`(rec_{copy}), `deepCopy`($args_{copy}$)$\rangle$ to $file$
  end
end

Figure 4-3: Pseudo-code for the ReCrash technique. If an exception is uncaught by the program (main), ReCrash stores the exception and a deep copy of the current shadow stack to the output file. The technique uses two auxiliary functions: `deepCopy` which copies all the reachable state of an object, and `copy` which is a parameter to this technique. The exact semantics of `copy` depends on the chosen copy strategy (see Section 4.1.2).

stack and the rest of the argument state uses references to the original objects (that might get side-effected). The different copying strategies, in order of increasing overhead, are:

reference (depth-0) Copying only the reference to the argument.

shallow (depth-1) Copying the argument itself. Each of the fields in the copy contains whatever the original argument did, a primitive or a reference. Shallow copying is resilient to direct side effects on the top-level primitive fields of the arguments.

depth-i Copying an argument to a specified depth: all the state reachable with i or fewer dereferences.

deep copy Copy the entire state. This strategy gives ReCrash the best chance of reproducing the same method execution, since it preserves the object state at the time of the method entry.

ReCrash has an additional copying option, **used-fields**, applicable to all copying strategies except reference. When the used-fields option is selected, ReCrash performs that select copying on arguments and on the fields of arguments that are used (read or written) in the method. For example, suppose that ReCrash is using shallow (depth-1) copying with used-fields, `x` is an argument to a method `m`, and `x.a` is used in `m`. Then ReCrash will perform shadow copying on `x` and `x.a`. The used fields are the fields that the method is likely to depend on, and therefore copying them increases the chance of reproducing the failure.

It is possible to use different strategies for different arguments. For instance, using shallow copy for the receiver, and using reference copy for the rest of the arguments. ReCrash always uses the reference strategy for immutable parameters (calculated using the technique presented in Chapter 5). A method's parameter p is immutable if the method never changes the state reachable from p. Therefore, an object passed to an immutable parameter will have the same state at the method entry and at the time of the failure.

4.1.2.2 Monitoring Fewer Methods

ReCrash need not monitor methods that are unlikely to expose or reveal problems, or that cannot be used in the generated tests. Those include empty methods, non-public methods, and simple methods such as getters and setters.

second-chance mode It would be most efficient to monitor only methods that will fail. However, it is impossible to compute this set of methods in advance. One way of approximating this set is to record a set of methods that already failed, updating the set each time a new method fails.

This is the underlying idea behind *second chance*, a mode of operating ReCrash that only monitors methods that have contributed to a failure at least once. In second chance mode, ReCrash initially monitors no method calls. Each time a failure occurs, ReCrash enables method argument monitoring for all methods found on the (real) stack back-trace at the time of the failure.

This mode is efficient, but requires a failure to be repeated twice (possibly with other failures in between) before ReCrash can reproduce it. Second chance mode has no impact on the reproducibility of a failure (the second time the failure appears).

4.1.2.3 Which Optimizations to Use

When using ReCrash, a developer needs to decide which optimizations to use. Which copying strategy should ReCrash use for the arguments? Should ReCrash use the used-fields option on the arguments? Should ReCrash use the second-chance mode?

The answers depend on the developer's needs and the subject program. For example, if the developer doesn't mind missing the first time a failure happens, and the failure occurs relatively often, the second chance mode is a good fit. If the developer wants ReCrash to reproduce all possible failures and can suffer the performance hit, then deep copy might be the right mode. We found that using the copying strategy shallow copying (depth-1) with used-fields enabled ReCrash to reproduce most failures with acceptable performance overhead.

4.1.3 Test Generation Phase

The test generation phase of ReCrash attempts to generate a unit test for each method invocation $\langle id, \mathtt{m}, rec_{copy}, args_{copy} \rangle$ on the ReCrash shadow stack. The pseudo-code for the test generation phase is presented in Figure 4-4.

ReCrash generates a test for each of the method frames in the shadow stack (s). ReCrash restores the state of the arguments (modulo the shadow stack) that were passed to m in execution id, and then invokes m the same way it was invoked in the original execution. Only tests that end with the same exception as the original failure are saved. Storing more than one test that ends with the same failure is useful because it is possible that some tests reproduce a failure, but would not help the developer understand, fix, or check her solution. See Section 4.2.2 for an example of such a case.

4.2 Implementation

We have implemented the ReCrash technique for Java. This section describes the implementation, ReCrashJ, using as an example the program in Figure 4-5. ReCrashJ has two phases: monitoring (Section 4.2.1) and test generation (Section 4.2.2). Section 4.2.3 discusses implementation details of the optimizations of Section 4.1.2.

4.2.1 Implementation of the Monitoring Phase

ReCrashJ instruments an existing program (in Java class file format, using the ASM instrumentation tool [105]) to perform the monitoring phase of the ReCrash technique (Section 4.1.1). The instrumented program can be deployed in the field instead of the original program. Figure 4-6 shows the instrumented version of the program in Figure 4-5.

The instrumentation has four parts, one for each of the tasks in Figure 4-3.

on program start The shadow stack is implemented using static fields in the ShadowStack class.

When the program starts, the requested copy strategies for the receiver and the arguments

Input: *inFile*- containing deep copy of the shadow stack
Output: *outFile*- containing tests reproducing the failure

```
begin
    e : exception, the original exception resulting in the failure
    testSuite : Collection of test sources, each test reproducing the original failure
    outFile : output file for the generated tests
    testSuite ← empty test suite
    e ← load exception from inFile
    len ← get length of shadow stack from inFile

    for i =1 to len do
        t : test source
        t ← generateTest(inFile, i)
        if execution of t ends with e then
            add t to testSuite
        end
    end
    store testSuite into outFile
end

generateTest(File file, int i)
begin
    ⟨id, m, rec_copy, args_copy⟩ ← pop i^th tuple from file
    output("test m.id")
    output("rec = load i^th receiver from file")
    output("args = load i^th arguments from file")
    output("rec.m(args)");
end
```

Figure 4-4: Pseudo-code for ReCrash test generation. ReCrash generates a test for every method on the original stack at the time of the failure, using the saved copy of the arguments. ReCrash outputs each test whose executions results in the same exception. The `generateTest` subroutine creates the source for a test. Each test loads the appropriate receiver and arguments from the file containing the shadow stack, and then calls the method on the receiver with the arguments.

```
1 class RandomCrash {
2   public String hexAbs(int x) {
3     String result = null;
4     if (x > 0)
5       result = Integer.toHexString(x);
6     else if (x < 0)
7       result = Integer.toHexString(-x);
8     return toUpperCase(result);
9   }
10
11  public String toUpperCase(String s) {
12    return s.toUpperCase();
13  }
14
15  public static void main(String args[]) {
16    RandomCrash rCrash = new RandomCrash();
17    rCrash.hexAbs(random.nextInt());
18  }
19 }
```

Figure 4-5: This program, taken from [143], will crash with a null pointer exception in the `toUpperCase` method, when the argument `x` to the method `hexAbs` is `0`. Since the value of `x` is randomly chosen (line 17), this crash is not deterministically repeatable. Figure 4-6 presents the instrumented version of the program.

is set (lines 33 and 34). The different copy strategy classes implement the strategies of Section 4.1.2.1.

on entry to method `m` The receiver and the arguments to the method are stored on the shadow stack at the beginning of each method. In addition, ReCrashJ generates an id for the invocation of `m`. The id is stored as a local variable (*id*) in the method. This is demonstrated by lines 3–5 and 18–20 of Figure 4-6. The specific behavior of the methods `addReceiver` and `addArgument` is determined by the type of the ShadowStack (see Section 4.2.3 for more details). ReCrash always uses shallow copy for immutable parameters (Chapter 5).

on non-exceptional exit from method `m` If the method successfully returns without a crash, ReCrashJ removes all the data (arguments and receiver) about the method execution from the shadow stack. ReCrashJ uses the unique identifier *id* to perform this cleanup (lines 13, 23).

on top-level uncaught exception In order to react to a thrown exception that is not caught by main, ReCrashJ replaces the original main by a new main as shown in lines 27–37 in Figure 4-6. The new main invokes the original main in a try-catch block and handles exceptions. When an exception is caught by the new try-catch block (line 36), ReCrashJ serializes the information on the shadow stack, and stores it to the output file (line 36).

```
1  class RandomCrash {
2    public String hexAbs(int x) {
3      int _id = ShadowStack.push("hexAbs");
4      ShadowStack.addReceiver(this);
5      ShadowStack.addArgument(x);
6      String result = null;
7      if (x > 0)
8         result = Integer.toHexString(x);
9      else if (x < 0)
10        result = Integer.toHexString(-x);
11
12     String ret = toUpperCase(result);
13     ShadowStack.popUntil(_id);
14     return ret;
15   }
16
17   public String toUpperCase(String s) {
18     int _id = ShadowStack.push("toUpperCase");
19     ShadowStack.addReceiver(this);
20     ShadowStack.addArgument(s);
21
22     String ret = s.toUpperCase();
23     ShadowStack.popUntil(_id);
24     return ret;
25   }
26
27   public static void __original_main__(String args[]){
28     RandomCrash rCrash = new RandomCrash();
29     rCrash.hexAbs(random.nextInt());
30   }
31
32   public static void main(String args[]) {
33     ShadowStack.setReceiverCopyingStrategy(new ShallowCopy());
34     ShadowStack.setArgumentsCopyingStrategy(new ReferenceCopy());
35     try  __original_main__(args);
36       catch (Throwable e)  StackDataWriter.writeStackData(e);
37   }
38 }
```

Figure 4-6: The instrumented program of Figure 4-5. Instrumentation code is bold.

4.2.1.1 Serialization

To serialize objects and store the serialized objects into a file, ReCrashJ uses the XStream framework [6] rather than Java serialization. Java serialization is limited to classes implementing the `java.io.Serializable` interface, and in which all fields must be of `Serializable` types. XStream does not have this limitation. ReCrash should be similarly applicable to any language with a library for marshalling/unmarshalling data. It may be an advantage of Java that Java has libraries that represent external resources, such as files, as Java objects. It may be that a language like Eiffel that uses opaque pointers would be at a relative disadvantage.

4.2.1.2 Alternatives to the Shadow Stack

The instrumented program is deployable. No other program or configuration is needed in order to run it. Both the Java Platform Debugger Architecture (JPDA) [2] and the Java Virtual Machine Tool Interface (JVMTI) [3] provide features to access Java objects in the stack with low overhead. However, in order to use these tools, we would have to deploy a separate program (in addition to the instrumented program) to communicate with either JPDA or JVMTI.

4.2.1.3 Reproducing Non-Crashing Failures

A developer may wish to reproduce failures other than crashes, for example exceptions that are caught by an exception handler or errors that do not result in an exception. In this case, the vendor can add calls to `writeStackData` wherever the program becomes aware of a failure—for example, in a catch-all handler or where an invariant is found to be false in an invariant validation routine.

As an example, consider the method `processList` in Figure 4-7. This method processes large lists. It first tries to minimize the input list by removing duplicates before processing it (call in line 2). However, if due to a bug, the minimization method fails and returns `null`, it is possible to process the entire list without minimization. Thus, the processing method is able to recover from the bug in the minimization method and continue. However, the developer will probably be interested in debugging the problem in the minimization method. In this case, the developer can signal to ReCrash (using the annotation in Line 6) that it should reproduce the state of the method in this case.

4.2.2 Implementation of the Test Generation Phase

ReCrashJ uses the stored shadow stack to generate a JUnit `testSuite`. Figure 4-8 shows the tests that ReCrashJ generated for the crash in Figure 4-5. Figure 4-2 shows two of the tests generated for the Eclipse compiler crash reported in Figure 4-1. Each test in the suite loads the receiver and the method arguments for one method from the serialized shadow stack, and then calls that method, which results in the same exception as the original crash. To facilitate debugging, for each argument (including receiver) that has a custom `toString` method, ReCrashJ writes the argument's string representation as a comment (lines 16, 21, 32, 36 of Figure 4-2).

```
1  public void processList(List inputList) {
2      List minimizedList = minimizeList(inputList);
3
4      // minimizedList should not be null
5      if (minimizedList == null) {
6          StackDataWriter.writeStackData(); // record this state
7          minimizedList = inputList;
8      }
9
10     ...
11 }
```

Figure 4-7: A manually written ReCrash annotation helps record a program state and reproduce errors that do not results in uncaught exceptions.

Not every object is dynamically read from the stored shadow stack when a test is executed. ReCrashJ writes the values of primitives, strings, and null objects directly into the tests. For example, see lines 8 and 18 of Figure 4-8.

4.2.2.1 Generating Multiple Tests for Each Crash

The tests in Figure 4-8 demonstrate a reason to create multiple tests that reproduce the crash, one for each method on the stack. The first test in Figure 4-8 is useless in detecting and solving the problem, because the developer is unable to understand the source of the null argument. This test would also continue to fail even when the problem is solved. On the other hand, the second test captures a value that is not handled correctly by the `hexAbs` method. This test is useful in determining and verifying a solution for the problem.

4.2.2.2 Extra Information

When instrumenting a subject program, developers can embed an identifier, such as a version number, in the subject program. This identifier will appear in the generated test cases, as shown in line 2 of Figure 4-2. This identifier can help the developers to identify which version of their program failed.

4.2.3 Optimizations

In order to implement the different copying strategies of Section 4.1.2.1, ReCrashJ uses the Java Cloneable interface. ReCrashJ automatically adds the clone method to all classes that do not already implement it. The added clone method copies primitive fields and the references of non-primitive fields.

ReCrashJ uses a simple static points-to analysis to determine the set of fields used in each method.

```
1 public void test_toUpperCase() {
2   ShadowStackReader.setMethodStackItem(2);
3
4   // load receiver
5   RandomCrash rec = (RandomCrash)ShadowStackReader.readReceiver(0);
6
7   // Method invocation
8   rec.toUpperCase(null);
9 }
10
11 public void test_hexAbs() {
12   ShadowStackReader.setMethodStackItem(1);
13
14   // load receiver
15   RandomCrash rec = (RandomCrash)ShadowStackReader.readReceiver(0);
16
17   // Method invocation
18   rec.hexAbs(0);
19 }
```

Figure 4-8: Tests generated by ReCrash for the program of Figure 4-5.

ReCrashJ approximates simple methods (i.e., getters and setters) as methods with no more than six opcodes. We use six opcodes as the bound since the standard getter (`getX(){returnx}`) has 4 opcodes and the standard setter (`setX(intx){this.x = x}`) has 6 opcodes without debug information.

4.3 Experimental Study

We evaluated ReCrashJ by performing experiments with crashes of real applications. We designed the experiments around the following research questions:

Q1 How reliably can ReCrashJ reproduce crashes?

Q2 What is the size of the stored deep copy of the shadow stack?

Q3 Are the tests generated by ReCrashJ useful for debugging?

Q4 What is the overhead (time and memory) of running ReCrashJ?

Our results indicate that ReCrashJ can reproduce many crashes, that it generates useful tests, that it incurs low overhead, and that the size of the stored data (serialized shadow stack) is small. We present the analysis of two real crashes in detail and show that the tests generated by ReCrashJ help to locate the source of the problem. In addition, the developers of one subject program found the generated test cases helpful. Overall, the experimental results indicate ReCrashJ is effective, scalable, and useful.

program	crash name	exception type	# frames shadow stack	# of reproducible tests reference	used-fields	copy	size of shadow stack(kb)
Javac	j1	null pointer	17	5	5	5	374
	j2	illegal argument	23	11	11	11	448
	j3	null pointer	8	1	1	1	435
	j4	index out of bounds	28	11	11	11	431
SVNKit	s1	index out of bounds	3	3	3	3	36
	s2	null pointer	2	2	2	2	34
	s3	null pointer	2	2	2	2	33
Eclipsec	e1	null pointer	13	0	1	8	62
BST	b1	class cast	3	3	3	3	5
	b2	class cast	3	3	3	3	5
	b3	unsupported encoding	3	3	3	3	25

Figure 4-9: Subject programs and crashes used in our experimental study. For each crash, ReCrashJ generates multiple test cases that aim to reproduce the original crash. In the used-fields column ReCrashJ used the shallow (depth-1) copying strategy with the used-fields option. The serialized shadow stack size is in gzipped KB.

4.3.1 Subject Systems and Methodology

We use the following subject programs in our experiments:

- **Javac-jsr308**[1] is the OpenJDK Java compiler, extended with the implementation of JSR308 ("Annotations on Java Types") [58]. We used four crashes that were provided to us by the developers. Javac-jsr308 version 0.1.0 has 5,017 methods in 86 kLOC.
- **SVNKit**[2] is a subversion[3] client. We used three crash examples for SVNKit bug reports #87 and #188. SVNKit version 0.8 has 2,819 methods in 22 kLOC.
- **Eclipsec**[4] is a Java compiler included in the Eclipse JDT. We used the crash from bug #30280 found in the Eclipse bug database. In version 2.1 of Eclipse, Eclipsec has 4,700 methods in 83 kLOC.
- **BST**[5] is a toy subject program used by Csallner in evaluating CnC [39, 40]. We used three BST crashes found by CnC. BST has 10 methods in 0.2 kLOC.

We used the following experimental procedure. We ran the ReCrashJ-instrumented versions of the subject programs on inputs that made the subject programs crash. We counted how many test

[1] http://groups.csail.mit.edu/pag/jsr308/
[2] http://svnkit.com/
[3] http://subversion.tigris.org
[4] http://www.eclipse.org
[5] http://www.cc.gatech.edu/cnc/index.html

cases reproduced each crash. We repeated this process for the different argument copying strategies introduced in Section 4.1.2.1, and with and without the second chance mode of Section 4.1.2.2. For the required parameter immutability classification (see Section 4.1.2.2) ReCrashJ runs PIDASA (Chapter 5 one time for each subject program. It took less than half an hour to calculate the parameter immutability classification for each of the subject programs.

4.3.2 Reproducibility

Q1 How reliably can ReCrashJ reproduce crashes?

ReCrashJ was able to reproduce the crash in all cases (Figure 4-9). For some crashes (b1, b2, b3, s1, s2, and s3), every candidate test case reproduces the crash. For other crashes (e1, j1, j2, j3, and j4), only a subset of the generated test cases reproduces the crash.

In most cases, simply copying references is enough to reproduce crashes ('reference' column). However, in some cases an argument is side-effected, between the method entry and the point of the failure in the method, in such a way that will prevent the crash if the modified argument had been supplied on the method entry. In those cases (e.g., e1), using at least the shallow copying strategy with used-fields (Section 4.1.2.1) was necessary to reproduce the crash.

4.3.3 Stored Deep Copy of the Shadow Stack Size

Q2 What is the size of the stored deep copy of the shadow stack?

If ReCrashJ is deployed in the field, and a crash happens, the user will need to send the serialized deep copy of the shadow stack to the program developers. For each crash, the last column of Figure 4-9 presents the size of the serialized deep copy of the shadow stack for each of the inspected crashes. The size can be further reduced if the tests are generated and evaluated locally. In this case ReCrash could trim the data from frames whose candidate test cases were discarded, or perform other minimization of the test cases.

4.3.4 Usability Study

Q3 Are the tests generated by ReCrashJ useful for debugging?

To learn whether ReCrashJ's test cases help developers to find errors, we analyzed the generated test cases for each crash. We present a detailed analysis of two crashes, e1 and j4. We also present developers' comments about the utility of the generated tests for j1-4.

Eclipsec bug (e1): Figure 4-12 presents the bug resulting in crash e1. Eclipsec crashes in method `canBeInstantiated` (line 19) because an earlier if statement in lines 9–11 failed to set the variable `hasError` to `true`. Using the generated tests (Figure 4-2) the developer would have been led to the buggy code. The developer would fix this problem by adding `hasError = true` on line 11. Notice that the test case for method `canBeInstantiated` (not shown in the figure) will reproduce the crash, but is not helpful in understanding it as the state of the parameter is already

corrupted. Also, note that just looking at the backtrace does make the problem obvious: stepping through will be more useful (the error is far removed from the crash, but is in a method on the stack). This is an example of why it is important to generate tests for multiple methods on the stack.

Javac-jsr308 bug (j4): Using Javac-jsr308 to compile source code containing an annotation with too many arguments results in an index-out-of-bounds exception. Figure 4-10 shows the erroneous source code. The compiler assumes that the `parameters` and `arguments` lists are of the same size (line 9), but they might not be.

```
1 public Void visitMethodInvocation (MethodInvocationTree node, Void p) {
2   List<AnnotatedClassType> parameters = method.getAnnotatedParameterTypes();
3
4   List<AnnotatedClassType> arguments =  new LinkedList<AnnotatedClassType>();
5   for (ExpressionTree arg : node.getArguments())
6       arguments.add(factory.getClass(arg));
7
8   for (int i = 0; i < arguments.size(); i++) {
9     if (!checker.isSubtype(arguments.get(i), parameters.get(i)))
10        ...
11  }
12 }
```

Figure 4-10: A code snippet that illustrates a Javac-jsr308 crash (j4). The crash (in line 9) happens when the parameters list is shorter then the arguments list.

ReCrashJ generates multiple test cases that reproduce the crash; one test is shown in Figure 4-11.

Note that the generated test does not require the whole source code and encodes only the necessary minimum to reproduce the crash. This makes ReCrashJ especially useful in scenarios where the failure happens in the field, and the user cannot provide the data (possibly proprietary) for debugging.

Developer testimonials We gave the tests for j1-4 to two Javac-jsr308 developers and asked for comments about the tests' usefulness. We received positive responses from both developers.

Developer 1: "I often have to climb back up through a stack trace when debugging. ReCrash seems to generate a test method for multiple levels of the stack, which would make it useful." "I find the fact that you wouldn't have to wait for the crash to occur again useful."

Developer 2: "One of the challenging things for me in debugging is that when I set a break point, the break point maybe be executed multiple times before the actual instance where the error is cased, [...] Using ReCrash, I was able to jump (almost directly) to the necessary breakpoint."

```
1 public void test_visitMethodInvocation() throws Throwable {
2   // load receiver
3   // rec = ...
4   SubtypeVisitor rec = (SubtypeVisitor)
5                           ShadowStackReader.readReceiver(0);
6
7   // load arguments
8   // arg_1 = test("foo", "bar", "baz");
9   MethodInvocationTree arg_1 = (MethodInvocationTree)
10                                   ShadowStackReader.readArgument(1);
11
12    // Method invocation
13   rec.visitMethodInvocation(arg_1, null);
14 }
```

Figure 4-11: Test case generated for a Javac-jsr308 crash (j4).

```
1 public class QualifiedAllocationExpression {
2   public TypeBinding resolveType(BlockScope scope) {
3     TypeBinding receiverType = null;
4     boolean hasError = false;
5     if (anonymousType == null) {
6       if ((enclosingInstanceType =
7            enclosingInstance.resolveType(scope)) == null){
8         hasError = true;
9     } else if (enclosingInstanceType.isArrayType()) {
10       ...
11         //hasError = true; Missing and causing the error
12     } else if ((this.resolvedType =
13                 receiverType = ...)) == null) {
14         hasError = true;
15     }
16     ...
17     // limit of fault-tolerance
18     if (hasError) return receiverType;
19     if (!receiverType.canBeInstantiated()) ...
20     ...
21   }
22 }
```

Figure 4-12: Buggy source code from Eclipsec (Eclipse Java compiler) causing bug #30280. The program crashes with a `NullPointerException` in the `canBeInstantiated` method (line 19). This case happens when the positive path of the `if` in line 9 is taken, in which case `hasError` is not set to false (line 11).

task	execution time			
	original	**reference**	**shallow w/used-fields**	**deep copy**
SVNKit checkout	1.17	1.62 (38%)	1.75 (50%)	1657 (142,000%)
SVNKit update	0.56	0.62 (11%)	0.63 (13%)	657 (118,000%)
Eclipsec Content	0.95	1.08 (13%)	1.12 (15%)	114 (12,000%)
Eclipsec String	1.07	1.36 (27%)	1.39 (31%)	1656 (155,000%)
Eclipsec Channel	1.27	1.72 (34%)	1.74 (37%)	8101 (638,000%)
Eclipsec JLex	3.45	4.93 (42%)	5.51 (60%)	> 2 days

task	second-chance execution time			
	original	**reference**	**shallow w/used-fields**	**deep copy**
SVNKit checkout	1.17	1.17 (0.0%)	1.18 (0.8%)	1.42 (21%)
SVNKit update	0.56	0.56 (0.0%)	0.56 (0.3%)	0.56 (0.8%)
Eclipsec Content	0.95	0.97 (1.5%)	0.96 (0.9%)	3.98 (317%)
Eclipsec String	1.07	1.09 (1.7%)	1.08 (0.8%)	8.99 (742%)
Eclipsec Channel	1.27	1.27 (0.1%)	1.27 (0.0%)	16.6 (1,210%)
Eclipsec JLex	3.45	3.47 (0.7%)	3.48 (1.1%)	1637 (47,000%)

Figure 4-13: Execution times of the original and instrumented programs in seconds. Slowdowns from the baseline appear in parentheses. The columns are described in Section 4.1.2.1.

4.3.5 Performance Overhead

Q4 What is the runtime overhead (time and memory) of ReCrashJ?

We compared the time and memory usage of the original and instrumented versions of the subject programs, while performing the following tasks (the three Java classes were taken from the `/samples/nio/server` directory in JDK 1.7):

SVNKit checkout:	Checking out a project [6], 880 files, 44Mb
SVNKit update:	Updating a project, 880 files, 44Mb
Eclipsec Content:	Compiling `Content.java` (48 LOC)
Eclipsec String:	Compiling `StringContent.java` (99 LOC)
Eclipsec Channel:	Compiling `ChannelIOSecure.java` (642 LOC)
Eclipsec JLex:	Compiling JLex version 1.2.4 (7,841 LOC) [7]

Figure 4-13 compares the execution time of the original subject programs and the instrumented ones, measured using the UNIX `time` command. Because of variability in network, disk, and CPU usage, there is some noise in the measurements, but the trend is clear. The deep copy version is completely unusable, except possibly in second chance mode, where it might be usable for in-house testing. A more optimized implementation could be more efficient, but will probably still be impractical.

[6] `http://amock.googlecode.com/svn/trunk`

[7] `http://www.cs.princeton.edu/~appel/modern/java/JLex/`

Even copying only the references can be expensive, 11%–42% run-time overhead. The shallow copying strategy with used-fields has a very similar overhead, 13%–60% overhead. These values are probably usable for in-house testing.

Second chance mode, however, is where our system shines. It reduces the overhead to a barely noticeable 0%–1.7% — and that is *after* a crash has already been observed, before which the overhead is negligible (essentially 0%). Second chance mode obtains these benefits by monitoring only a very small subset of all the methods in the program. This simple idea is sufficient, effective, and efficient.

task	maximum memory usage (MB)		
	original	**shallow w/used-fields**	**overhead%**
SVNKit checkout	8.7	9.3	6.9
SVNKit update	7.6	7.8	2.6
Eclipsec Content	4.8	8.4	75.0
Eclipsec String	5.3	9.5	79.2
Eclipsec Channel	5.2	9.9	90.3
Eclipsec JLex	9.1	11.1	21.9

Figure 4-14: Memory use overhead of the instrumented programs, with shallow and used-fields copying strategy, as measured by JProfiler.

For the memory usage comparison, we used JProfiler[8] to measure the maximum memory used in performing each task. Figure 4-14 shows the memory usage — in our experiments, ReCrashJ adds 0.2M–4.7M memory overhead for the effective shallow copying strategy with the used-fields option. The memory overhead for the same copying strategy in the second-chance mode was negligible.

4.4 Discussion

This section discusses limitations of our technique and threats to the validity of our experiments.

4.4.1 Limitation in Reproducing Failures

ReCrash cannot necessarily reproduce all failures. The following list contains some causes that might prevent ReCrash from reproducing a failure:

failures dependent on timing This category of failures includes concurrency-related failures such as failures resulting from a race condition in a multi-threaded application. Monitoring concurrency timing dependent failures and reproducing them is future work.

failures dependent explicitly on external resources ReCrash may be unable to reproduce a failure that depends on external resources such as file system, network, hardware devices, etc.

[8] http://www.ej-technologies.com/products/jprofiler/overview.html

For example, a failure might depend on loading a file that no longer exists. ReCrash could be combined with tracing tools for calls that access external resources. This would be less expensive than performing full tracing.

failures dependent implicitly on external resources Failures may depend implicitly on external resources. For instance, an argument such as a socket or GUI object cannot be serialized.

failures dependent on global state, or side-effected argument state Failures may depend on unserialized state. Or a failure might depend on a part of the argument state that is not stored to the shadow stack (due to the copying strategy) and is side-effected between the method entrance and the time of the failure.

We believe that not storing global state is not a great limitation for Java, as noted in the beginning of the chapter and validated by our experiments. Evaluating the effect of storing global state on reproducibility is future work.

ReCrash might still be able to reproduce a failure that depends on one of these causes. This may occur if the program cannot handle some state that is caused from one of this scenarios, but the state it is recorded in the ReCrash serialized data. In this case the developer might be able to find, understand, and fix the failure even without understanding the exact condition that led to the state exposing it.

4.4.2 Buggy Methods Not in the Stack at Time of the Failure

```
1 public class SVNCommandLine {
2   private list myURLs;
3
4   public String getURL(int index) {
5     return (String) myURLs.get(index);
6   }
7
8   protected void init(String[] arguments) throws SVNException {
9     myURLs = new ArrayList();
10    for(int i = 0; i < arguments.length; i++) {
11      ...
12    }
13  }
14 }
```

Figure 4-15: A code snippet from SVNKit illustrating crash s1. The method causing the crash (`init`) will not be on the stack at the time of the crash.

It is possible that a failure is caused by a method that is not on the call stack at the time of the failure. For example, consider the code in Figure 4-15. This figure contains the source of crash s1

(SVNKit bug #87). An index-out-of-bounds exception is thrown if the user omits the URL from the check-out command. When no URL is supplied, the method `init` should set a default URL or throw an exception about a missing URL, but it does neither.

The program in Figure 4-15 will crash on Line 5, and at the time of the crash the call stack will contain the method `getURL` but will not contain the method `init`. ReCrash generates a test case for each frame in the call stack. The test case ReCrash creates for method `getURL` calls the method `getURL` with 0 as a parameter, and the exception is thrown. This test does not help the developer to find the reason for the illegal state of `myURLs`. ReCrash will also generate a test for the `run` method (not shown), which calls `init`. Debugging using the test for the `run` method will expose the real bug.

4.4.3 Reusing ReCrash-Generated Tests

The ReCrash-generated tests are not intended to be added as is to the program test suite. The reason is that the tests de-serialize objects. If the structure of a class changes, the de-serialization will stop working. However, the generated test cases provide the skeleton of tests, and developers can replace the de-serialization with normal code.

Developers are not intended to examine the JUnit code of the generated tests to find a bug. Rather, a developer can use a debugger to examine the test's execution, revealing the cause of the failure.

4.4.4 Privacy

One of the problems with debugging deployed applications is that users may be reluctant to send data to the developers for fear of exposing proprietary data. For instance, a user who discovers a bug in Eclipse might be unable to send proprietary source files to the developer.

While we have not solved this problem, using ReCrash might be seen as an intermediate solution. Sending only the parts of the shadow stack that are used in selected tests is less likely to contain proprietary data. In addition, it should be possible for a client to prevent ReCrash from storing sensitive data by marking it (i.e., via program annotations). Another possible solution is to provide a shadow stack reader to the client, so that the client can review the encoded data and decide which parts of it (or which tests) are safe to send to the developers.

4.4.5 Threats to Validity

We identify the following key threats to validity.

Systems and crashes examined might not be representative. We might have a system or crash selection bias. Our experiments use every crash we considered. We did not discard any data that was not reproducible to us, nor did we find a crash that ReCrash could not reproduce. However, we examined only 11 crashes from 4 subject systems. It is time-consuming to find a real bug (by studying bug reports), download an older version of the software, compile it,

and reproduce the bug. We may have accidentally chosen systems or crashes that have better (or worse) than average ReCrash reproducibility.

All failures are runtime exceptions. Our experiments use failures that manifest as runtime exceptions, such as null pointer or index out of bounds exceptions. However, ReCrashJ can monitor user-annotated exceptions or errors as discussed in Section 4.2.1.3. Evaluating the usefulness of the ReCrash annotation requires a user study. Future experiments and user studies should consider other types of failures including user-annotated non-exception points.

4.5 Related Work

4.5.1 Record and Replay

Many existing record and replay techniques for postmortem analysis and debugging [31,32,44,53, 64,90,102,160] are based on three components: checkpoints, logging, and replay. The checkpoint provides a snapshot of the full state of the program at specific times, while the log records events between snapshots. The replay component uses the log to replay the events between checkpoints. By contrast, ReCrash performs a checkpoint at each method entry and has no log of events, only an in-memory record of stack elements. ReCrash does not replay the entire execution, instead ReCrash allows the developer to observe the system in several states before the failure, then run the original program until the failure is reproduced. ReCrash's logging simplicity allows it to be deployed remotely with relatively low overhead. Most of the previous techniques are designed for in-house testing or debugging, and have unreasonable overhead for deployed applications.

BugNet [102], ReVirt [53], and FlashBack [136] require changes to the host operating system while FDR [160] uses a proprietary hardware recorder. ReCrash, on the other hand, can be deployed in any environment, and can be used to reproduce a recorded failure in different environments.

Choi et al. [31], Instant Replay [90], BugNet [103], and many others emphasize the ability to deterministically replay an execution of a non-deterministic program. They are able to reproduce race conditions and other non-deterministic failures. In order to achieve this goal, these techniques either impose a large space and time overhead [31,90], or they only allow replaying a narrow window of time [103]. Similar to BugNet [103], ReCrash only allows replaying a part of the execution before the failure. ReCrash is only able to deterministically reproduce a non-deterministic failure if one of the generated tests captures the state after the non-determinism. ReCrash could be combined with other monitoring tools (storing only some of the environment dependencies) to create an efficient technique that can reproduce non-deterministic failures.

jRapture [137], test factoring [129], test carving [55], and ADDA [32] capture the interactions between the program and the environment to create smaller test cases or enable reproducing failures. These techniques capture a trace, and then run the subject code in a special harness, such as a mock object representing all interactions with the rest of the system, that reads values out of the trace whenever the subject code would have interacted with its environment. Test factoring does

this at the level of method calls; ADDA does it at the level of file operations and C standard library functions. By contrast, our approach does not record a trace; it sets up the system in a particular start state, and then the system runs unassisted. However, ReCrash can be viewed as writing mock objects in the method call data. The objects recorded are a faithful representation of the actual objects down to a given depth. At lower depths their fields contain values that are possibly out of date, but that in practice are effective in reproducing program behavior.

4.5.2 Test Generation

Contract Driven Development [91] generates test cases using contracts (pre- and post-conditions) written by developers. If a contract is violated during the execution of an application in debug mode, CDD captures the transitive state of the arguments to one method (often the failing method). CDD then attempts to use the captured state to generate a test case that reproduces the violation. Since the arguments' state is captured after the violation, it is equal to ReCrash's reference copying strategy. ReCrash might generate more useful tests and reproduce more failures because ReCrash can store arguments' state before a violation occurs, and because ReCrash generates multiple tests, one for each method on the shadow stack at the time of the failure. CDD is designed to be used in the development process, whereas ReCrash can be used either in-house or in-field. ReCrash monitors the stack and generates a test case without the need for special IDE support.

CnC [40], JCrasher [39], Eclat [110], Randoop [111], and DSDCrasher [41] use random inputs to find program crash points. Their generated tests may not be representative of actual use, whereas ReCrash generates tests that reproduce real crashes. Chapter 2 presented Palulu, a model-based test generation tool that similarly attempt to generates tests based on values observed during an actual program execution. However, the test generate by Palulu are used for regression testing, and therefor attempt to exercise similar but different functionality. The tests generate by ReCrash attempt to imitate the exact original execution to reproduce the problem.

4.5.3 Remote Data Collection

Crash reporting systems such as Dr. Watson [4], Apple's Crash Reporter [1], and Talkback [5] send a stack back-trace or program core-dump to the vendor. The vendor uses the stack back-trace to correlate the report to known problems. If several reports share similar characteristics, the vendor may try to reproduce the original failure. However, reproducing the original failure requires non-trivial human effort. The core-dump provides one snapshot of the program state at the end of time (after the crash). ReCrash provides several partial snapshots and enables execution of each of them, using a test. ReCrash's stored data is smaller than a core-dump and thus is easier to send to the vendor.

4.6 Conclusion

We have introduced the a test generation technique ReCrash that helps maintainers eliminate bugs by generating unit tests that reproduce software failures. The tests utilize partial snapshots of the

program state that ReCrash captures on each method execution and records in the case of a failure. ReCrash is simple to implement, it is scalable, and it generates simple, helpful test cases that effectively reproduce failures. Our ReCrashJ tool implements the technique for Java.

We have evaluated ReCrashJ with real crashes from Javac-jsr308, SVNKit, Eclipsec, and BST. ReCrashJ created tests that reproduced every crash we have inspected, and developers found the generated tests useful for debugging. The performance overhead of programs instrumented by ReCrashJ was 13%–64% for the shallow copying strategy with used-fields and 0%–1.7% for all copying strategies in the second-chance mode. In our experiments, ReCrashJ increases memory usage when using the effective shallow copying strategy with used-fields, by 0.2M–4.7M. The size of the stored snapshots, the serialized shadow stack, is manageable: 0.5k—448k. ReCrashJ is usable in real software deployment.

The ReCrashJ tool described in this chapter is publicly available for download from `http://pag.csail.mit.edu/ReCrash`.

Chapter 5

Reference Immutability Inference

This chapter presents the use of combined dynamic and static analysis, to solve the problem of finding parameter references that can only be accessed in a read-only way. This problem is traditionally solved using only static analysis. We proposed to combine a dynamic analysis with a simple and scalable static analysis to present a scalable and efficient solution.

The read-only information is used in two of our techniques. It is used to improve upon the efficiency of the ReCrash technique(Chapter 4) by reducing ReCrash's overhead in copying arguments on method entry. It is used to reduce the sizes of models created by Palulu (Chapter 2) to increase the amount of model state explored in the generation of the regression tests. In this chapter we present a formal definition of parameter mutability, and an analysis for classifying the mutability of parameters. Such definition was not previously presented in the literature.

Knowing which method parameters are accessed in a read-only way, and which ones may be mutated, is useful in many software engineering tasks, such as modeling [24], verification [28,142], compiler optimizations [33, 130], program transformations such as refactoring [62], test input generation [10], regression oracle creation [95, 155], invariant detection [59], specification mining [43], program slicing [140, 153], and program comprehension [48,51].

Unintended mutation is a common source of errors in programs. Such errors can be difficult to discover and debug, because their symptoms often appear far from the location of the error, in time or in space. The problem of unintended mutation is exacerbated by the difficulty of expressing the programmer's design intent. A type system or a program annotation system offers an approximation to the truth regarding whether particular mutations can occur. The advantage of such approximations is that they are checkable. However, every statically checkable approximation prevents certain references, that are never used for mutation, from being so annotated because the proof of that fact is beyond the capability of the program analysis. A trivial example is that whether a given mutating statement can be executed is undecidable, but the problem is not a theoretical one and arises frequently in practice [19, 112, 146, 164]

Informally, reference immutability guarantees that a given reference is not used to modify its referent. (An immutable reference is sometimes known as a "read-only reference".) This definition has been the basis of much previous research [19, 22, 49, 77, 86, 131, 134, 145, 164]. The informal description is intuitively understandable, but because it is vague, different people have different intuitions. For example, "used" might refer to the time the reference is in scope [12, 131] or the

entire execution [22, 145]. The referent might be considered to be only a single object [22, 138], or also all objects (transitively) referred to by subfields [86, 146, 164]. Modification might refer to the referent's concrete state [12, 22, 131] or to its abstract state [145, 164]. Modifications via aliases may be ignored [131] or counted as a mutation [164]. (See Section 5.8.2 for further discussion of related work.)

A precise definition of parameter reference mutability is a prerequisite to understanding, evaluating, or verifying an algorithm or tool. Some research [49, 86, 104, 134] informally describes reference mutability but does not define the concept precisely. Capabilities [22], Javari [145], and IGJ [164] use type/annotation systems to formally define different variants of reference immutability. Such formal definitions (e.g., type rules) conservatively approximate what mutations can actually occur: some references, that cannot be used to perform mutation, are mutable according to the definition. By contrast, our definition is precise: it does not depend on any particular implementation or approximation. Another difference is that the previous definitions considered mutations in the entire program execution when determining method parameter mutability. In our definition, which is inspired by [12, 131], only mutations during the execution of the containing method are counted when determining parameter mutability. However, our definition easily extends to the definitions used by Javari, Capabilities, and IGJ. (Section 5.1.3 discusses the extension).

The lack of a precise, formal definition of reference (im)mutability has made it difficult to determine both *whether* a program analysis tool correctly approximates the ideal, and *how closely* it does so. In addition, previous approaches for computing mutability suffered from scalability problems (static systems) and accuracy problems (dynamic systems). Our research addresses these issues. We formally define parameter (im)mutability; we present Pidasa, an approach to detect mutability that combines the strengths of both static and dynamic analysis resulting in a system with better run-time performance and accuracy. Finally, we qualitatively and quantitatively compare both our definition and our implementation to existing immutability inference systems.

The Pidasa approach to mutability detection combines the strengths of static and dynamic analyses. Previous work has employed static analysis techniques to detect *immutable* parameters. Computing accurate static analysis approximations threatens scalability, and imprecise approximations can lead to weak results. Dynamic analyses offer an attractive complement to static approaches, both in not using approximations and in detecting *mutable* parameters. In our approach, different analyses are combined in stages, forming a "pipeline", with each stage refining the overall result.

This chapter focuses on reference immutability of parameters. Parameter immutability is an important and useful special case of reference immutability, and it is supported by tools against which we can compare our definition. Our definition extends in a straightforward way to general reference immutability.

Parameter reference immutability can be computed per method implementation. A type system for reference immutability may enforce that method overriding preserve mutability of parameters, and type annotations can be easily computed from the per-implementation reference immutability information.

Contributions. This chapter makes the following contributions:

- A formalization of a widely-used definition of parameter reference immutability that does not depend on a type or annotation system.
- The first staged analysis approach for discovering parameter mutability. Our staged approach is unusual in that it combines static and dynamic stages and it explicitly represents analysis incompleteness. The framework permits both sound and unsound analyses for immutability. We examine the tradeoffs involved in such a choice.
- Mutability analyses. Our novel dynamic analysis scales well, yields accurate results (it has a sound mode as well as optional heuristics), and complements other immutability analyses. We extend the dynamic analysis with random input generation, which improves the analysis results by increasing code coverage. We also explore a new point in the space of static techniques with a simple but effective static analysis.
- Evaluation. We have implemented our framework and analyses for Java, in a tool Pidasa. We performed two kinds of experiments.

  The first kind of experiments, investigates the costs and benefits of various sound and unsound techniques, including both Pidasa and that of Sălcianu and Rinard [131]. Our results show that a well-designed collection of fast, simple analyses can outperform a sophisticated analysis in both run-time performance and accuracy.

  The second kind of experiments demonstrates that the our formal definition is useful in exposing similarities and differences between approaches to immutability. We find that the evaluated analyses (Pidasa, JPPA [131], Javari [19, 146], and JQual [69]) produce results that are very close to our formal definition, which confirms the need for a common formal base.

Outline. The remainder of this chapter is organized as follows. Section 5.1 formally defines parameter reference immutability and illustrates it on an example (Appendix 5.2 contains a longer example). Section 5.3 presents our staged mutability analysis framework. Sections 5.4 and 5.5 describe the new dynamic and static mutability analyses that we developed as components in the staged analysis. Section 5.6 experimentally evaluates various instantiations of the staged analysis framework. Section 5.7 experimentally compares results computed by the Pidasa inference tool and other existing tools to the formal definition. Section 5.8 surveys related work, and Section 5.9 concludes.

5.1 Parameter Mutability Definition

Informally, an object pointed by an immutable parameter reference cannot be changed. However, there exist different interpretations of what is considered a change exist. For instance, the change might be to the set of reachable references from the parameter (deep), or to the references immediately reachable from the parameter (shallow). The change might happen during the method call, or after the method call. A change might happen through an aliasing reference, without even referring

to the parameter in the method (reference vs. object immutability). Thus, a formal definition of parameter reference mutability is non-trivial.

Reference immutability differs from object immutability, which states that a given object cannot be mutated through any reference whatsoever. Both immutability variants have their own benefits. Reference immutability is more useful for specifying that a procedure may not modify its arguments, even if the caller reserves the right to do so before or after the procedure call. Object immutability is more useful, for example, when optimizing a program to store constant data in a shared section or read-only memory. Reference immutability is more fine-grained—reference immutability may be combined with an alias and escape analysis to infer object immutability [19, 131], but not vice-versa.

Section 5.1.1 informally defines parameter mutability. Section 5.1.2 demonstrates the intuition on an example. Section 5.1.3 formalizes parameter mutability. Section 5.1.4 and Appendix 5.2 give examples of the formal definition.

5.1.1 Informal definition

Parameter p of method m is reference-*mutable* if there exists any execution of m during which p is *used* to mutate the state of the object pointed to by p. Parameter p is said to be *used* in a mutation, if the left hand side of the mutating assignment was p or was obtained from p via a series of field accesses and copy operations during the given execution. (Array accesses are analogous. The index expression is not considered to be used; that is, `a[b.f].f = 0` is not a mutation of `b`.) The mutation may occur in m itself or in any method that m transitively calls. The state of an object o is the part of the heap that is reachable from o by following references, and includes the values of reachable primitive fields. Thus, reference immutability is deep—it covers the entire abstract state of an object, which includes the fields of objects reachable from the object.

If no such execution (in all possible well-typed invocations of the method) exists, the parameter p is reference-*immutable*.

For example, in the following method:

```
void f(C c) {
  D d = c.d;
  E e = d.e;
  e.f = null;
}
```

parameter `c` is used in the mutation in the last statement since the statement is equivalent to `c.d.e.f = null`.

The definition presented in this section, as well as its formalism in Section 5.1.3, is perfectly precise, but it is not computable, due to the quantification over all possible executions. Any tool can only infer an approximation of this definition of reference-mutability. (By contrast, some other attempts at defining mutability are based on a computable algorithm, but that does not capture the actual behavior of the program.) Section 5.7 compares several mutability inference tools, including our Pidasa tool (Section 5.3), to the definition.

5.1.2 Immutability Classification Example

In the code in Figure 5-1, parameters p1 – p5 are reference-*mutable*, since there exists an execution of their declaring method such that the object pointed to by the parameter reference is modified *via the reference*. Parameters p6 and p7 are reference-*immutable*.
***mutable* parameters**:

- p1 may be directly modified in modifyParam1 (line 8).
- p2 is passed to modifyParam1, in which it may be mutated.
- p3 is *mutable* because line 19 modifies p3.next.next.
- p4 is directly modified in modifyAll (line 18), because the mutation occurs via reference p4.
- p5 is *mutable* because the state of the object passed to p5 can get modified on line 19 via p5. This can happen because p4 and p3 might be aliased: for example, in the call modifyAll(x2, x2, x1, false). In this case, the reference to p5 is copied into c and then used to perform a modification on line 19. Neither p3 nor p5 is considered reference mutable as a result of line 18 even though they might be aliased to p4 at run-time. I.e., the mutating assignment is equal to p5.next = null.

***immutable* parameters**:

- p6 and p7 are reference-*immutable*. No execution of either method doNotModifyAnyParam or doNotModifyAnyParam2 can modify an object passed to p6 or p7.

5.1.3 Formal Definition

We present our formal parameter mutability definition in three steps. The first step defines a core object-oriented language (Section 5.1.3.1). The second step defines the evaluation rules (Section 5.1.3.2) (i.e., the operational semantics) that compute whether a parameter reference is used in a mutation. The third step (Section 5.1.3.3) formally defines parameter reference-mutability (Definition 1) by adding universal quantification over all possible well-typed invocations.

Our definition, described in this section, is quantified over all well-typed expressions. As a consequence, our definition does not depend on any specific execution, although it is defined using operational semantics in Section 5.1.3.1.

5.1.3.1 Core Language

We define reference-mutability in the context of Mut Lightweight Java (MLJ), an augmented version of Lightweight Java (LJ) [145], a core calculus for Java with mutation. LJ extends Featherweight Java (FJ) [80] to include field assignment with the set construct. To support the field

```
1  class C {
2     public C next;
3  }
4
5  class Main {
6     void modifyParam1(C p1, boolean doIt) {
7        if (doIt) {
8           p1.next = null;
9        }
10    }
11
12    void modifyParam1Indirectly(C p2, boolean doIt) {
13       modifyParam1(p2, doIt);
14    }
15
16    void modifyAll(C p3, C p4, C p5, boolean doIt) {
17       C c = p3.next;
18       p4.next = p5;
19       c.next = null;
20       modifyParam1Indirectly(p3, doIt);
21    }
22
23    void doNotModifyAnyParam(C p6) {
24       if (p6.next == null)
25          System.out.println("p6.next␣is␣null");
26    }
27    void doNotModifyAnyParam2(C p7) {
28       doNotModifyAnyParam(p7);
29    }
30 }
```

Figure 5-1: Example code that is used to illustrates parameter immutability. All non-primitive parameters other than p6 and p7 are *mutable*.

assignment, LJ adds the store (S) that records a mapping from each object to its class and field record. LJ uses the field record ($\mathcal{F}$) to record a mapping from each field to the values they contain.

The syntax of MLJ is presented in Figure 5-2. The syntax of MLJ is identical to the syntax of LJ without the parts of LJ that are related to generics and wild-cards. Those parts are irrelevant to reference immutability. Control flow constructs such as `if` only propagate, never introduce, mutation, so adding them to MLJ would only clutter the presentation. Similarly, arrays offer no more insight than field accesses. Other features, such as local variables, can be emulated in MLJ. The definition ignores reflection, but some tools handle it, including the dynamic analyses of our Pidasa tool (Section 5.3).

$$
\begin{array}{lcll}
\mathtt{Q} & ::= & \mathtt{class\ C\ extends\ C\ \{\overline{C\ f};\ K\ \overline{M}\}} & \textit{class declarations} \\
\mathtt{K} & ::= & \mathtt{C(\overline{C\ f})\ \{super(\overline{f});\ this.\overline{f} = \overline{f};\}} & \textit{constructor declarations} \\
\mathtt{M} & ::= & \mathtt{C\ m(\overline{C\ x})\{\ return\ e;\}} & \textit{method declarations} \\
\mathtt{e} & ::= & \mathtt{x} & \textit{expressions} \\
 & | & \mathtt{e.f} & \\
 & | & \mathtt{e.m(\overline{e})} & \\
 & | & \mathtt{new\ C(\overline{e})} & \\
 & | & \mathtt{set\ e.f = e\ then\ e} & \\
\mathtt{C} & & \textit{class names} & \\
\mathtt{m} & & \textit{method names} & \\
\mathtt{f} & & \textit{field names} & \\
\mathtt{x} & & \textit{parameter names} &
\end{array}
$$

Figure 5-2: Syntax of Mut Lightweight Java: same as Lightweight Java [145] without generics and wild cards. $\overline{X}$ is a vector of X. The set syntax was chosen to avoid the complications of allowing multiple expressions within a method. Method arguments and assignment expressions are evaluated left-to-right.

Mut Lightweight Java allows us to define mutability in the context of the underlying structure of Java, without being overwhelmed by the complexity of the full language. Since the definition is quantified over all possible executions, it does not depend on any specific execution, even though it uses operational semantics mechanism.

To avoid clutter, in the description below we present the operational semantics (evaluation) rules, but omit the typing rules, which are irrelevant to reference immutability.

Figure 5-3 shows the notations of MLJ that are used in the operational semantics, with the changes from LJ shown shaded. A value $v = (\!|o, \mathcal{M}|\!)$ in MLJ is a pair containing the corresponding value o from LJ, and a mutability set $\mathcal{M}$. If the value v can be modified, then the formal parameters in $(\!|o, \mathcal{M}|\!)$ are parameter reference-mutable. The *ret* expression provides a hook for removing formal parameters from mutability sets when a method exits. A *ret* expression cannot be written by a user, but is created during the evaluation of a program. The mutability store (Ω) contains the set of parameters that have been discovered to be reference-mutable.

r	::=	$ret\ \mathtt{m}^i\ \mathtt{e}$	*return expressions*
$\mathcal{M}$	::=	$\emptyset$	*mutability sets*
	|	$\{C.m^i.x, \ldots\}$	*parameters*
v	::=	$(\!\| o, \mathcal{M} \|\!)$	*values : pairs of object, mutability set*
S	::=	$\emptyset$	*stores*
	|	$S, o = (C, \mathcal{F})$	*object bindings*
$\mathcal{F}$	::=	$\emptyset$	*field records*
	|	$\mathcal{F}, \mathtt{f} = v$	*field bindings*
Ω	::=	$\emptyset$	*mutability stores*
	|	$\{C.m.x, \ldots\}$	*mutable parameters*
o			*objects*
i			*natural numbers*

Figure 5-3: Mut Lightweight Java notations used in the operational semantics (Figure 5-4). Changes from Lightweight Java [145] are shaded.

5.1.3.2 Evaluation Rules

In MLJ, evaluation maintains a mutability set $\mathcal{M}$ for each value v. $\mathcal{M}$ contains a set of parameters that were used in obtained the value. A formal parameter $C.\mathtt{m}.\mathtt{x}$ is added to set $\mathcal{M}$ for a given value v if either v was the value bound to parameter $C.\mathtt{m}.\mathtt{x}$ or if v was a result of a dereference operation from a value that had $C.\mathtt{m}.\mathtt{x}$ in its mutability set. On exit from the ith invocation of $\mathtt{m}$, evaluation removes from all mutability sets, any parameters added on the corresponding method entry. A modification to a value causes the parameters in the value's mutability set to be classified as mutable.

Figure 5-4 shows the operational semantics rules for MLJ. Each reduction rule is a relationship, $\langle \mathtt{e}, S, \Omega \rangle \longrightarrow \langle \mathtt{e}', S', \Omega' \rangle$, where expression $\mathtt{e}$ with store S and mutability store Ω reduces to expression $\mathtt{e}'$ with store S' and mutability store Ω'. The Ω is only updated when a parameters is assigned. In this case the rules add all the parameters that were used to obtain the mutated reference to the Ω. The changes from LJ, in computation and congruence rules, are shaded in Figure 5-4. The congruence rules remain essentially unchanged between LJ and MLJ. We describe each computation rule, and the additional computation in MLJ.

[R-Field] This rule evaluates field accesses.

LJ: locates the value v_2 stored in the field and returns it.

MLJ: adds the mutability set $\mathcal{M}_{v_1}$, of the dereferenced value v_1, to the mutability set of v_2. The field access causes v_2 to be accessed from the parameters in $\mathcal{M}_{v_1}$, as well as the parameters that were previously used to obtain v_2 (i.e., the mutability set of v_2).

[R-Invk] This rule evaluates method invocations.

Computation:

$$\frac{\mathcal{S}(o_1) = \langle \mathtt{C}, \mathcal{F}\rangle \qquad \mathcal{F}(\mathtt{f}_i) = v_2 \qquad v_1 = (\!|o_1, \mathcal{M}_{v_1}|\!) \qquad v_2 = (\!|o_2, \mathcal{M}_{v_2}|\!)}{\langle v_1.\mathtt{f}_i,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle\ (\!|o_2, \mathcal{M}_{v_1} \cup \mathcal{M}_{v_2}|\!)\ ,\ \mathcal{S}\ ,\ \Omega\ \rangle} \quad \text{[R-FIELD]}$$

$$\frac{\mathcal{S}(o) = \langle \mathtt{C}, \mathcal{F}\rangle \quad mbody(\mathtt{m},\ \mathtt{C}) = \overline{\mathtt{x}}.\mathtt{e}_0 \quad v = (\!|o, \mathcal{M}_v|\!) \quad i = invocations(m) + 1 \quad \overline{v} = (\!|\overline{o}, \mathcal{M}_{\overline{v}}|\!)}{\langle v.\mathtt{m}(\overline{v}),\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle\ ret\ m^i\ [\ (\!|\ \overline{o}\ ,\overline{\mathcal{M}_v} \cup \mathtt{C}.\mathtt{m}^i.\overline{\mathtt{x}}|\!)\ /\overline{\mathtt{x}},\ (\!|\ o\ ,\mathcal{M}_v \cup \{C.m^i.\mathtt{this}\}|\!)\ /\mathtt{this}]\mathtt{e}_0,\ \mathcal{S}\ ,\ \Omega\ \rangle} \quad \text{[R-INVK]}$$

$$\frac{v = (\!|o, \mathcal{M}_v|\!)}{\langle ret\ \mathtt{m}^i\ v,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle (\!|o, remove(\mathtt{m}^i, \mathcal{M}_v)|\!),\ removeAll(\mathtt{m}^i, \mathcal{S})\ ,\ \Omega\ \rangle} \quad \text{[R-RET]}$$

$$\frac{o \notin \mathrm{dom}(\mathcal{S}) \qquad \mathcal{F} = [\overline{\mathtt{f}} \mapsto \overline{v}]}{\langle \mathtt{new}\ \mathtt{C}(\overline{v}),\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle\ (\!|\ o\ ,\varnothing|\!)\ ,\ \mathcal{S}[o \mapsto \langle \mathtt{C},\ \mathcal{F}\rangle]\ ,\ \Omega\ \rangle} \quad \text{[R-NEW]}$$

$$\frac{\mathcal{S}(o_1) = \langle \mathtt{C}, \mathcal{F}\rangle \qquad v_1 = (\!|o_1, \mathcal{M}_{v_1}|\!)}{\langle \mathtt{set}\ v_1.\mathtt{f}_i = v_2\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{e}_b,\ \mathcal{S}[o_1 \mapsto \langle \mathtt{C},\ \mathcal{F}[\mathtt{f}_i \mapsto v_2]\rangle]\ ,\ \Omega \cup removeSuperscript(\mathcal{M}_{v_1})\ \rangle} \quad \text{[R-SET]}$$

Congruence:

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{e}_0',\ \mathcal{S}'\ ,\ \Omega'\ \rangle}{\langle \mathtt{e}_0.\mathtt{f},\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{e}_0'.\mathtt{f},\ \mathcal{S}'\ ,\ \Omega'\ \rangle} \quad \text{[RC-FIELD]}$$

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{e}_0',\ \mathcal{S}'\ ,\ \Omega'\ \rangle}{\langle \mathtt{e}_0.\mathtt{m}(\mathtt{e}),\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{e}_0'.\mathtt{m}(\mathtt{e}),\ \mathcal{S}'\ ,\ \Omega'\ \rangle} \quad \text{[RC-INVK-RECV]}$$

$$\frac{\langle \mathtt{e}_i,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{e}_i',\ \mathcal{S}'\ ,\ \Omega'\ \rangle}{\langle v.\mathtt{m}(\overline{v}, \mathtt{e}_i, \overline{\mathtt{e}}),\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle v.m(\overline{v}, \mathtt{e}_i', \overline{\mathtt{e}}),\ \mathcal{S}'\ ,\ \Omega'\ \rangle} \quad \text{[RC-INVK-ARG]}$$

$$\frac{\langle \mathtt{e}_i,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{e}_i',\ \mathcal{S}'\ ,\ \Omega'\ \rangle}{\langle \mathtt{new}\ \mathtt{C}(\overline{v}, \mathtt{e}_i, \overline{\mathtt{e}}),\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{new}\ \mathtt{C}(\overline{v}, \mathtt{e}_i', \overline{\mathtt{e}}),\ \mathcal{S}'\ ,\ \Omega'\ \rangle} \quad \text{[RC-NEW-ARG]}$$

$$\frac{\langle \mathtt{e}_0,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{e}_0',\ \mathcal{S}'\ ,\ \Omega'\ \rangle}{\langle \mathtt{set}\ \mathtt{e}_0.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{set}\ \mathtt{e}_0'.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}'\ ,\ \Omega'\ \rangle} \quad \text{[RC-SET-LHS]}$$

$$\frac{\langle \mathtt{e}_v,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{e}_v',\ \mathcal{S}'\ ,\ \Omega'\ \rangle}{\langle \mathtt{set}\ v.f = \mathtt{e}_v\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{set}\ v.f = \mathtt{e}_v'\ \mathtt{then}\ \mathtt{e}_b,\ \mathcal{S}'\ ,\ \Omega'\ \rangle} \quad \text{[RC-SET-RHS]}$$

$$\frac{\langle \mathtt{e}_v,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle \mathtt{e}_v',\ \mathcal{S}'\ ,\ \Omega'\ \rangle}{\langle ret\ \mathtt{m}^i\ \mathtt{e}_v,\ \mathcal{S}\ ,\ \Omega\ \rangle \longrightarrow \langle ret\ \mathtt{m}^i\ \mathtt{e}_v',\ \mathcal{S}'\ ,\ \Omega'\ \rangle} \quad \text{[RC-RET]}$$

Figure 5-4: Operational semantics (evaluation rules) for Mut Lightweight Java. Changes from Lightweight Java [145] are shaded.

LJ: works in two stages. First, it finds the correct method m (using auxiliary function *mbody* which returns a pair $\overline{\mathtt{x}}$.e where $\overline{\mathtt{x}}$ are m's formal parameters and e is m's body). Second, it replaces the call with the body of m and replaces each formal parameter with the actual parameter.

MLJ: updates the mutability set of each formal parameter with the corresponding superscripted (with number of invocations) formal parameter, e.g., $m^i.x$. This rule also adds a `ret` expression call. The `ret` expression enables the [R-RET] rule to remove the same superscripted (with number of invocations) formal parameters from all the relevant mutability sets when the method exits. This rule uses the auxiliary method *invocations* which returns the number of times the method m was invoked (also ensures that i is fresh).

[R-RET] This rule evaluates `ret` expressions and it does not exist in LJ.

MLJ: removes superscripted (see rule [R-INVK]) method formal parameters from all mutability sets. The function *remove* removes the parameters of $\mathtt{m}^i$ from the mutability set of the returned value v. The function *removeAll* does the same for all other values in the store.

[R-NEW] This rule evaluates object allocations.

LJ: creates a newly-allocated object.

MLJ: sets the mutability set of the newly-allocated object o to the empty mutability set.

[R-SET] This rule evaluates `set` expressions, i.e., field writes.

LJ: updates the value stored in a field for a given object.

MLJ: adds all the un-superscripted parameters (auxiliary method `removeSuperscript`) from the mutability set of the modified object to the mutability store Ω.

5.1.3.3 Reference-Immutability Definition

We define reference-immutable and reference-mutable parameters:

Definition 1. ***(parameter reference mutability)*** *Parameter* x *of method* m *of class C is* reference-mutable *if there exists an expression* e *such that* $C.\mathtt{m.x} \in \Omega$ *at the end of the evaluation of* e. *Otherwise, p is* reference-immutable.

To simplify the presentation in the rest of this section, we will refer to $C.m.x$ as x and to $C.m$ as m. Theorem 1 demonstrates the properties of Definition 1 and its equivalence to the intuitive informal definition. An auxiliary Definition 2 formalizes the notion of a series of dereferences needed for Theorem 1.

Let $\phi(i, m, e)$ be the evaluation of the ith invocation (dynamically) of m in e. We write it as ϕ when the parameters can be inferred from context.

Let $v_x^\phi = (\!|o, \mathcal{M}_v \cup \mathtt{m}^i.\mathtt{x}|\!)$ be the value passed to x at the start of ϕ (Rule [R-INVK]).

Definition 2. *We inductively define $\delta_j^{\phi}(x)$, the executed dereference set of x during the first j evaluation steps of ϕ.*

$$
\begin{array}{lcll}
\delta_1^{\phi}(x) & = & \{v_x^{\phi}\} & \\
\delta_j^{\phi}(x) & = & \delta_{j-1}^{\phi}(x) \cup (\!|o_2, \mathcal{M}_{v_1} \cup \mathcal{M}_{v_2}|\!) & \textit{if the } j^{\text{th}} \textit{step is [R-FIELD] and } v_1 \in \delta_{j-1}^{\phi}(x) \\
\delta_j^{\phi}(x) & = & \delta_{j-1}^{\phi}(x) & \textit{otherwise}
\end{array}
$$

Let $\delta^{\phi}(x)$ be the set of executed dereferences at the end of ϕ.

Theorem 1. *A parameter x is reference-mutable iff there exist v, i, m, e, j such that $v \in \delta_j^{\phi}(x)$ and [R-SET] is evaluated on v.f during ϕ.*

Before proving Theorem 1 we present two auxiliary lemmas. Lemma 1 states that $m^i.x$ can exist in a mutability set only during ϕ. Lemma 2 states that if a value v is in the executed dereferences of a parameter x, then v's mutability set contains x.

Lemma 1. *Suppose that $v = (\!|o, \mathcal{M}_v|\!)$ and $m^i.x \in \mathcal{M}_v$. Such a value v can exist only during ϕ.*

Proof. $m^i.x$ is created and added to a mutability set in rule [R-INVK] when starting the evaluation ϕ. [R-RET], which evaluates at the end of ϕ, removes all parameters of the form $m^i.x$ from all mutability sets. □

Lemma 2 presents the equality of the mutability set and the set of "used" references.

Lemma 2. *Let $v = (\!|o, \mathcal{M}_v|\!)$. Then $m^i.\mathrm{x} \in \mathcal{M}_v$ iff $v \in \delta^{\phi}(x)$.*

Proof. $\longleftarrow$ Proof by induction on j where $v = v^j$ such that $v^j \in \delta_j^{\phi}(x) \wedge v^j \notin \delta_{j-1}^{\phi}(x)$.

If $j = 1$ then $v = v_x^{\phi}$ and $\mathtt{m}^i.x \in v_x^{\phi}$ by definition.

Otherwise, from Definition 2

$$
v^j = (\!|o_2, \mathcal{M}_{v_1} \cup \mathcal{M}_{v_2}|\!) \wedge v_1 \in \delta_{j-1}^{\phi}(x).
$$

By the induction assumption $\mathtt{m}^i.\mathrm{x} \in \mathcal{M}_{v_1}$. Thus $\mathtt{m}^i.\mathrm{x} \in \mathcal{M}_{v^j}$.

$\longrightarrow$ From $m^i.\mathrm{x} \in \mathcal{M}_v$ it follows that $\mathtt{m}^i.\mathrm{x}$ was added to $\mathcal{M}_v$ in either rule [R-INVK] or rule [R-FIELD] (the only two rules that augment a mutability set).

If $\mathtt{m}^i.\mathrm{x}$ was added to $\mathcal{M}_v$ in rule [R-INVK] then $v \in \delta_1^{\phi}(x)$.

If $\mathtt{m}^i.\mathrm{x}$ was added to $\mathcal{M}_v$ in rule [R-FIELD] then

$$
v = (\!|o_2, \mathcal{M}_{v_1} \cup \mathcal{M}_{v_2}|\!) \text{ and } \mathtt{m}^i.\mathrm{x} \in \mathcal{M}_{v_1} \text{ or } \mathtt{m}^i.\mathrm{x} \in \mathcal{M}_{v_2}.
$$

The rest of the proof follows by induction on the length of the executed [R-FIELD] from a value that contains $\mathtt{m}^i.\mathrm{x}$ in its mutability set. □

The proof of Theorem 1 follows directly from Lemma 2 and Lemma 1.

Proof. $\longrightarrow$ By Definition 1,

$$\exists \texttt{e} \text{ such that } \texttt{m.x} \in \Omega \text{ during the evaluation of } \texttt{e}.$$

Since rule [R-SET] is the only rule creating a larger Ω it follows that

$$\exists v = (\!| o, \mathcal{M}_v |\!) \text{ such that } \texttt{m}^i\texttt{.x} \in \mathcal{M}_v \text{ and [R-SET] is evaluated on } v.f.$$

By Lemma 1, v can only exist during ϕ. The proof now follows directly from Lemma 2.

$\longleftarrow$ By $v \in \delta_j^\phi(x)$ and Lemma 2,

$$m^i.\texttt{x} \in \mathcal{M}_v.$$

Since [R-SET] is evaluated on $v.f$ during ϕ, we get $\texttt{m}^i.\texttt{x} \in \Omega$. By Definition 1, x is mutable. □

Our definition accounts for mutations that occur during the dynamic extent of a method invocation. In some other mutability definitions [22, 145, 164], if a parameter's value escapes to another context in which the value is later mutated, then the parameter is mutable in the original method (e.g., a getter method). Our framework accommodates both varieties of definition: removing rule [R-RET] converts Definition 1 into the other variant. In the revised definition, a reference is not removed from the mutability set when its method exits.

The rest of this chapter uses *mutable* and *immutable* to refer to parameter reference mutable and parameter reference immutable, respectively.

5.1.4 Examples

We illustrate Definition 1 on two example functions in Figures 5-5 and 5-6. Each example contains a program in MLJ and a table with the evaluation of MLJ operational semantics on the program. Each line in the table presents an expression to be evaluated, the state of the stores corresponding to the expression, and the next rule to apply. Similarly to Featherweight Java [114], we treat Obj as a distinguished class name whose definition does not appear in the class table. Obj has an empty constructor, no methods, and no fields. In addition, irrelevant (to mutability) details such as calls to `super` in constructors and the `extends` keyword are omitted.

Example: Classifying a modified receiver as mutable (Figure 5-5)

The parameter `this` of the function `m` in Figure 5-5 is *mutable* because line 4 modifies `this.f`. The first two lines in the MLJ evaluation show the evaluation of the [R-NEW] rules. These rules creates the initial state of the store, and all the values created in them have empty mutability sets. The rule [R-INVK] is applied to the expression in step 3. Since it is the first invocation of the method m, parameters $\texttt{m}^1$ and $m^1.this$ are added to the mutability sets of the newly created values. The rule [R-SET] is applied to the expression in step 4. Since the object o_3 is modified, the parameters in its mutability set $\{m^1.this\}$ are added to the store of mutable parameters. Finally, the evaluation of the rule [R-RET] signals the exit from method m^1 and thus it removes all parameters superscripted by 1 from the mutability sets.

Example: Classifying aliased parameters (Figure 5-6)

```
1  class B {
2    Obj f;
3    B(){ this.f = new Obj() }
4    Obj m(Obj p){ return set this.f = p then p }
5  }
6  new B().m(new Obj())
```

	expression	S	Ω	next rule
1	$new\ B().m(new\ Obj())$	$\varnothing$	$\varnothing$	[R-NEW]
2	$\mathtt{new}\ B().m((\!|o_1, \varnothing|\!))$	$\{(o_1, \langle Obj, \varnothing\rangle)\}$	$\varnothing$	[R-NEW]
3	$(\!|o_3, \varnothing|\!).m((\!|o_1, \varnothing|\!))$	$\{(o_1, \langle Obj, \varnothing\rangle), (o_2, \langle Obj, \varnothing\rangle),$ $(o_3, \langle B, \{f : (\!|o_2, \varnothing|\!)\}\rangle)\}$	$\varnothing$	[R-INVK]
4	$ret\ \mathtt{m}^1\ set\ (\!|o_3, \{m^1.this\}|\!).f = (\!|o_1, \{m^1.p\}|\!)\ then\ (\!|o_1, \{m^1.p\}|\!)$	$\ldots$	$\varnothing$	[R-SET]
5	$ret\ \mathtt{m}^1\ (\!|o_1, \{m^1.p\}|\!)$	$\{(o_1, \langle Obj, \varnothing\rangle), (o_2, \langle Obj, \varnothing\rangle),$ $(o_3, \langle B, \{f : (\!|o_1, \{m^1.p\}|\!)\}\rangle)\}$	$\{m.this\}$	[R-RET]
6	$(\!|o_1, \varnothing|\!)$	$\{(o_1, \langle Obj, \varnothing\rangle), (o_2, \langle Obj, \varnothing\rangle),$ $(o_3, \langle B, \{f : (\!|o_1, \varnothing|\!)\}\rangle)\}$	$\{m.this\}$	*Done*

Figure 5-5: Classifying a mutable parameter. The Mut Lightweight Java program calls a method that updates a field in an object. The figure shows the evaluation of the program using the rules of Figure 5-4. After the rule [R-SET] is applied (step 4), the parameter `m.this` is classified as mutable. Symbol ... means that a store does not change between evaluation steps.

In function `m` (Figure 5-6), reference p_1 is *mutable* due to the modification in line 5. However, reference p_2 is reference-*immutable*—it is never used to make any modification to an object during the execution of `m`. The evaluation table in Figure 5-6 demonstrates that parameter p_2 of function `m` is not reference-*mutable* in the call `m(o, o)` (i.e., even when parameters are aliased). When the execution finishes, $m.p_2 \notin \Omega$ and thus it is not classified as mutable during the evaluation of the call `new B().n()` or any other call for that matter.

Our definition is concerned with *reference* mutability, which, together with aliasing information, may be used to compute object mutability. In the example of function `m` in Figure 5-6, the information that parameter p_2 is reference-immutable can be combined with information about p_1 and p_2 being aliased in the call `m(o, o)` to determine that, in that call, both objects may be modified.

Appendix 5.2 contains an additional, more involved, example of an evaluation of MLJ.

5.2 More Complicated Example of Reference Immutability

This section presents an additional example of Definition 1. We apply Definition 1 to the method `modifyAll` of Figure 5-1 (which is problematic for JPPA).

The left hand side of figure 5-7 contains the partial Java code for a simplified version of the method `modifyAll` of figure 5-1. The right hand side contains the same method after in MLJ.

It is obvious that parameters p_2 and p_3 of method `m` are reference-*mutable* (line 2 modifies p_2, and line 4 modifies p_3.f). It is harder to see that parameter p_1 is also reference-*mutable*. p_1 is mutated when, for example, parameters p_2 and p_3 are aliased (like in the call `m(x, y, y)`). In this case, the state of the object passed to p_1 is modified on line 4 using a series of dereferences

```
1  class B{
2    Obj f;
3    B(){ this.f = new Obj() }
4    Obj n(){ return this.m(this, this) }
5    B m(B p₁, B p₂){ return set p₁.f = new Obj() then this }
6  }

7  new B().n()
```

	expression	S	Ω	next rule
1	$new\ B().n()$	$\varnothing$	$\varnothing$	[R-New]
2	$(\!\|o_2, \varnothing\|\!).n())$	$\{(o_1, \langle Obj, \varnothing\rangle), (o_2, \langle B, \{f : (\!\|o_1, \varnothing\|\!)\}\rangle)\}$	$\varnothing$	[R-Invk]
3	$ret\ n^1\ (\!\|o_2, \{n^1.this\}\|\!).m((\!\|o_2, \{n^1.this\}\|\!), (\!\|o_2, \{n^1.this\}\|\!))$	...	$\varnothing$	[R-Invk]
4	$ret\ n^1\ ret\ m^1\ set\ (\!\|o_2, \{n^1.this, m^1.p_1\}\|\!).f =$ $new\ Obj()\ then\ (\!\|o_2, \{n^1.this, m^1.this\}\|\!)$	...	$\varnothing$	[R-New]
5	$ret\ n^1\ ret\ m^1\ set\ (\!\|o_2, \{n^1.this, m^1.p_1\}\|\!).f =$ $(\!\|o_3, \varnothing\|\!)\ then\ (\!\|o_2, \{n^1.this, m^1.this\}\|\!),$	$\{(o_1, \langle Obj, \varnothing\rangle), (o_2, \langle B, \{f : (\!\|o_1, \varnothing\|\!)\}\rangle)$ $(o_3, \langle Obj, \varnothing\rangle)\}$	$\varnothing$	[R-Set]
6	$ret\ n^1\ ret\ m^1\ (\!\|o_2, \{n^1.this, m^1.this\}\|\!)$	...	$n.this,\ m.p_1$	[R-Ret]
7	$ret\ n^1\ (\!\|o_2, \{n^1.this\}\|\!)$	...	$n.this,\ m.p_1$	[R-Ret]
8	$(\!\|o_2, \varnothing\|\!)$	...	$n.this,\ m.p_1$	

Figure 5-6: Classifying a mutable parameter while leaving another parameter, that points to a modified object, non mutable. Symbol ... means that a store does not change between evaluation steps. For the lack of local variables in the language, the utility method n sends the same object (this) as both parameters of the method m. The figure demonstrate how 2 parameters are classified as mutable, using the rules of Figure 5-4. After the rule [R-Set]is applied to the expression in line 5, the parameters $n.this$ and $m.p_1$ are classified as mutable. Parameter $m.p_2$ can not be in the modified set of any parameter in the body of method m and thus there is no execution that will modify $m.p_2$, and so $m.p_2$ is defined as immutable.

from $\mathtt{p}_1$. Sălcianu [130] presents this method as an example for unsoundness in his static analysis. Sălcianu's tool, JMH, wrongly classifies parameter $\mathtt{p}_1$ as immutable.

Figure 5-8 presents the application of Definition 1 to method m by evaluating m in MLJ. Step 2 contains the expression and the stores that are the results of evaluating all the initial constructors. Step 3 is the result of invoking the method n. Step 4 is the result of invoking the method m. Notably, $\mathtt{p}_2$ is replaced with $(\!|o_6, \{n^1.p_2, m^1.p_2\}|\!)$ and $\mathtt{p}_3$ is replaced with the same object (o_6) but with a different value $(\!|o_6, \{n^1.p_2, m^1.p_3\}|\!)$. After evaluating the first set expression (corresponding to $\mathtt{p}_2.\mathtt{f} = \mathtt{p}_1$), the parameters $\mathtt{n.p}_2$ and $\mathtt{m.p}_2$ in the mutability set of $(\!|o_6, \{n^1.p_2, m^1.p_2\}|\!)$ are classified as mutable. Step 6 presents the result of evaluating the field access. The resulting object o_4 has the combined mutability set of both its previous value $(\!|o_4, \{n^1.p_1, m^1.p_1\}|\!)$ (from the store) those of the

```
1   void m(B p1, C p2, C p3) {
2     p2.f = p1;
3     B l = p3.f;
4     l.f = this;
5   }
```

```
1   class A extends Object {
2     A(){ super(); }
3     A n(B p1, C p2){ return this.m(p1, p2, p2) }
4     A m(B p1, C p2, C p3){ set p2.f = p1 then q(p3.f) }
5     A q(B l){ return set l.f = this then this }
6   }
7   class B extends Object {
8     A f;
9     B(A f){ super(); this.f = f }
10  }

11  class C extends Object {
12    B f;
13    C(B f){ super(); this.f = f }
14  }

    new A().n(new B(new A()), new C(new B(new A())))
```

Figure 5-7: The left-hand side contains a simplified version of method `modifyAll` from Figure 5-1. Parameters p_2 and p_3 are mutable since they are mutated in lines 2 and 4. Parameter p_1 is also mutable. It will be mutated when parameters p_2 and p_3 are aliased, for example in the call m(x, y, y). The right-hand side contains the same method and the call m(x, y, y), converted to MLJ. Since MLJ has no local variables, we have converted the local variable l into a method's parameter (method q). Method n passes the same parameter twice to method m, used to implement the Java call m(x, y, y).

dereferenced value $(\!| o_6, \{n^1.p_2, m^1.p_3\} |\!)$. When that value is modified in step 7, all the parameters in its mutability set $(n^1.p_1, m^1.p_1, n^1.p_2, m^1.p_3)$ are classified as mutable. Thus definition recognizes that parameter $m.p_1$ is mutable.

5.3 Staged Mutability Analysis

The goal of any parameter-reference-mutability analysis is the classification of each method parameter (including the receiver) as either reference-mutable or reference-immutable.

In Pidasa approach, mutability analyses are combined in stages, forming a "pipeline". The input to the first stage is the initial classification of all parameters (typically, all *unknown*, though parameters declared in the standard libraries may be pre-classified). Each stage of the pipeline refines the results computed by the previous stage by classifying some *unknown* parameters. Once a parameter is classified as *mutable* or *immutable*, further stages do not change the classification. The output of the last stage is the final classification, in which some parameters may remain *unknown*.

Throughout this chapter, two objects are *aliased* if the intersection of their states contains at least one non-primitive object (the same object is reachable from both of them).

Our dynamic and static analyses complement each other to classify parameters in Figure 5-1

	expression	S	Ω	next rule
1	$new\ A().n(new\ B(new\ A()), new\ C(new\ B(\texttt{new}\ A())))$	$\emptyset$	$\emptyset$	[R-NEW]
2	$\llparenthesis o_1, \emptyset \rrparenthesis .n(\llparenthesis o_4, \emptyset \rrparenthesis, \llparenthesis o_6, \emptyset \rrparenthesis)$	$\{(o_1, \langle A, \emptyset \rangle), (o_2, \langle A, \emptyset \rangle), (o_3, \langle A, \emptyset \rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_2, \emptyset \rrparenthesis\} \rangle), (o_5, \langle B, \{f : \llparenthesis o_3, \emptyset \rrparenthesis\} \rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_5, \emptyset \rrparenthesis\} \rangle)\}$	$\emptyset$	[R-INVK]
3	$ret\ n^1\ \llparenthesis o_1, \{n^1.this\} \rrparenthesis .m(\llparenthesis o_4, \{n^1.p_1\} \rrparenthesis, \llparenthesis o_6, \{n^1.p_2\} \rrparenthesis, \llparenthesis o_6, \{n^1.p_2\} \rrparenthesis)$	...	$\emptyset$	[R-INVK]
4	$ret\ n^1\ ret\ m^1\ set\ \llparenthesis o_6, \{n^1.p_2, m^1.p_2\} \rrparenthesis .f =$ $\llparenthesis o_4, \{n^1.p_1, m^1.p_1\} \rrparenthesis\ then\ this.q(\llparenthesis o_6, \{n^1.p_2, m^1.p_3\} \rrparenthesis .f)$	...	$\emptyset$ $\emptyset$	[R-SET]
5	$ret\ n^1\ ret\ m^1\ this.q(\llparenthesis o_6, \{n^1.p_2, m^1.p_3\} \rrparenthesis .f)$	$\{(o_1, \langle A, \emptyset \rangle), (o_2, \langle A, \emptyset \rangle), (o_3, \langle A, \emptyset \rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_2, \emptyset \rrparenthesis\} \rangle), (o_5, \langle B, \{f : \llparenthesis o_3, \emptyset \rrparenthesis\} \rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_4, \{n^1.p_1, m^1.p_1\} \rrparenthesis\} \rangle)\}$	$n.p_2, m.p_2$	[R-FIELD]
6	$ret\ n^1\ ret\ m^1\ this.q(\llparenthesis o_4, \{n^1.p_2, m^1.p_3, n^1.p_1, m^1.p_1\} \rrparenthesis)$	...	...	[R-INVK]
7	$ret\ n^1\ ret\ m^1\ ret\ q^1\ set\ \llparenthesis o_4, \{n^1.p_2, m^1.p_3, n^1.p_1, m^1.p_1, q^1.p_1\} \rrparenthesis .f =$ $\llparenthesis o_1, \{n^1.this, m^1.this, q^1.this\} \rrparenthesis\ then\ \llparenthesis o_1, \{n^1.this, m^1.this, q^1.this\} \rrparenthesis$	...	...	[R-SET]
8	$ret\ n^1\ ret\ m^1\ ret\ q^1 \llparenthesis o_1, \{n^1.this, m^1.this, q^1.this\} \rrparenthesis$	$\{(o_1, \langle A, \emptyset \rangle), (o_2, \langle A, \emptyset \rangle), (o_3, \langle A, \emptyset \rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_1, \{n^1.this, m^1.this, q^1.this\} \rrparenthesis\} \rangle)$ $(o_5, \langle B, \{f : \llparenthesis o_3, \emptyset \rrparenthesis\} \rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_4, \{n^1.p_1, m^1.p_1\} \rrparenthesis\} \rangle)\}$	$n.p_2, m.p_2, m.p_3$ $n.p_1, m.p_1, q.p_1$	[R-RET]
9	$ret\ n^1\ ret\ m^1\ \llparenthesis o_1, \{n^1.this, m^1.this\} \rrparenthesis$	$\{(o_1, \langle A, \emptyset \rangle), (o_2, \langle A, \emptyset \rangle), (o_3, \langle A, \emptyset \rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_1, \{n^1.this, m^1.this\} \rrparenthesis\} \rangle)$ $(o_5, \langle B, \{f : \llparenthesis o_3, \emptyset \rrparenthesis\} \rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_4, \{n^1.p_1, m^1.p_1\} \rrparenthesis\} \rangle)\}$	$n.p_2, m.p_2, m.p_3$ $n.p_1, m.p_1, q.p_1$	[R-RET]
10	$ret\ n^1\ \llparenthesis o_1, \{n^1.this\} \rrparenthesis$	$\{(o_1, \langle A, \emptyset \rangle), (o_2, \langle A, \emptyset \rangle), (o_3, \langle A, \emptyset \rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_1, \{n^1.this\} \rrparenthesis\} \rangle)$ $(o_5, \langle B, \{f : \llparenthesis o_3, \emptyset \rrparenthesis\} \rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_4, \{n^1.p_1\} \rrparenthesis\} \rangle)\}$	$n.p_2, m.p_2, m.p_3$ $n.p_1, m.p_1, q.p_1$	[R-RET]
11	$\llparenthesis o_1, \emptyset \rrparenthesis$	$\{(o_1, \langle A, \emptyset \rangle), (o_2, \langle A, \emptyset \rangle), (o_3, \langle A, \emptyset \rangle)$ $(o_4, \langle B, \{f : \llparenthesis o_1, \emptyset \rrparenthesis\} \rangle)$ $(o_5, \langle B, \{f : \llparenthesis o_3, \emptyset \rrparenthesis\} \rangle)$ $(o_6, \langle B, \{f : \llparenthesis o_4, \emptyset \rrparenthesis\} \rangle)\}$	$n.p_2, m.p_2, m.p_3$ $n.p_1, m.p_1, q.p_1$	*Done*

Figure 5-8: Applying Definition 1 to the code in Figure 5-7. The figure shows the evaluation using the rules of Figure 5-4. Symbol ... means that a store does not change between evaluation steps. After the rule [R-SET]is applied in step 7, the parameter $m.p_1$ is classified as mutable. The mutation to this parameter is not obvious and it depends on the aliasing between the other two parameters p_2 and p_3.

into *mutable* and *immutable*, in the following steps:

1. Initially, all parameters are *unknown*.

2. A flow-insensitive, intra-procedural static analysis classifies p1, p4, and p5 as *mutable*. The analysis classifies p6 as *immutable*—there is no direct mutation in the method and the parameter does not escape.

3. An inter-procedural static analysis propagates the current classification along the call-graph. It classifies p2 as *mutable* since it is passed to an already known mutable parameter, p1. It also classifies parameter p7 as *immutable* since it can only be passed to *immutable* parameters.

4. A dynamic analysis classification of p3 depends on the given example execution. The dynamic analysis classifies p3 as *mutable* if a method (similar to the main method below)

   ```
   void main() {
     modifyAll(x1, x2, x2, false);
   }
   ```

   is supplied or generated (see Section 5.4.4). Otherwise, the dynamic analysis classifies p3 as *unknown*.

Our staged analysis correctly classifies all parameters in Figure 5-1. However, this example poses difficulties for purely static or purely dynamic techniques. On the one hand, static techniques have difficulties correctly classifying p3. This is because, to avoid over-conservatism, static analyses often assume that on entry to a method all parameters are fully un-aliased, i.e., point to disjoint parts of the heap. In our example, this assumption may lead such analyses to incorrectly classify p3 as *immutable* (in fact, Sălcianu uses a similar example to illustrate the unsoundness of his analysis [130, p.78]). On the other hand, dynamic analyses are limited to a specific execution and only consider modifications that happen during that execution. In our example, a purely dynamic technique may incorrectly classify p2 as *immutable* if during the execution, p2 is not modified.

Combining mutability analyses can yield an analysis that has better accuracy than any of the components. For example, a static analysis can analyze an entire program and can prove the absence of a mutation, while a dynamic analysis can avoid analysis approximations and can prove the presence of a mutation.

Combining analyses in a pipeline also has performance benefits—a component analysis in a pipeline may ignore previously classified parameters. This can permit the use of techniques that would be too computationally expensive if applied to an entire program.

The problem of mutability inference is undecidable, so no analysis can be both sound and complete. An analysis is *i-sound* if it never classifies a *mutable* parameter as *immutable*. An analysis is *m-sound* if it never classifies an *immutable* parameter as *mutable*. An analysis is *complete* if it classifies every parameter as either *mutable* or *immutable*.

In our staged approach, analyses may explicitly represent their incompleteness using the *unknown* classification. Thus, an analysis result classifies parameters into three groups: *mutable*, *immutable*, and *unknown*. Previous work that used only two output classifications [124, 126] loses information by conflating parameters/methods that are known to be mutable with those where analysis approximations prevent definitive classification.

Some tasks, such as many compiler optimizations [33, 130] require i-sound results (unless the results are treated as hints or are used online for only the current execution [159]). Therefore, we have i-sound versions of our static and our dynamic analyses. However, other tasks, such as test input generation [10], can benefit from more complete immutability classification while tolerating i-unsoundness. For this reason, we have devised several unsound approximations to increase the completeness (recall) of the analyses. Clients of the analysis can create an i-sound analysis by combining only i-sound components. Other clients, desiring more complete information, can use i-unsound components as well. Figure 5-9 summarizes the soundness characteristics of the analyses presented in this chapter.

Analysis	Name	Section	i-sound	m-sound
dynamic	D	5.4.2	✓*	✓
dynamic heuristic **A**	DA	5.4.3	-	✓
dynamic heuristic **B**	DB	5.4.3	✓*	-
dynamic heuristic **C**	DC	5.4.3	✓*	-
dynamic heuristics **A**,**B**,**C**	DH	5.4.3	-	-
static intraprocedural	S	5.5.2	✓	-
static intraprocedural heuristic	SH	5.5.2.1	-	-
static interproc. propagation	P	5.5.3	✓	✓†
JPPA [131]	J	5.6.2	-	✓*
JPPA + `main`	JM	5.6.2	-	✓*
JPPA + `main` + heuristic	JMH	5.6.2	-	-

Figure 5-9: The static and dynamic component analyses used in our experiments. "✓*" means the algorithm is trivially sound, by never outputting the given classification. "✓†" means the algorithm is sound but our implementation is not.

Some previous work [124, 126, 131] has used the term "sound" to denote i-sound while being m-sound by avoiding from classifying any mutable parameters or impure methods. [131] provide proofs of the i-sound property and optionally provides m-unsound mutability classifications in its output. *Mutable* parameters and/or impure methods classifications are useful even for applications that are using only *immutable* parameters and/or pure methods, since they allows a better sense of the recall achieved by the analysis.

Salcianu's algorithm is not i-sound: Consider the classification of the method `modifyAll().p3` of Figure 5-1. `p3` is reference-*mutable* according to definition 1 and object-*mutable* by Sălcianu's definition. However, Sălcianu's algorithm make the assumption that on top-level entry to a method all parameters are fully un-aliased (i.e., point to disjoint parts of the heap) since accounting for all possible side effects under all possible aliasing conditions would yield unusable conservative results in his static analysis. Therefore, in this case his algorithm will unsoundly classify `c1` as

immutable.

In the context of compiler optimizations, misclassifying a mutable parameter as *immutable* can lead to impermissible changes in behavior, but misclassifying *mutable* can only lead to lost optimization opportunities. This lead previous work to work under the assumption that a mutable parameter can never be classified as *immutable*. In certain contexts, such as many software engineering tasks, scalability and accuracy are key metrics of quality, so the assumptions made by previous work are unnecessarily restrictive. As one example, test suite generators should use methods with side effects in the bodies of test cases, and should use methods without side effects as "observers" in their assertions [10, 95, 155]. Misclassification is not fatal: if a method is mistakenly marked as side-effect-free, this merely reduces test suite coverage, and even if a supposed observer actually modifies some of its parameters, a test case can still be effective in revealing errors. Similar arguments apply to invariant detection and specification mining. Program understanding is yet another domain in which perfect classification is not essential, since humans easily handle some inaccuracy [101]. Human judgment is required in any event, because automatically generated results do not necessarily match programmer abstractions (e.g., automatic analyses do not ignore modifications to caches).

To address the needs of different mutability analysis contexts, the analyses presented in this chapter can be combined in pipeline with varying precision and recall properties. The client of the analysis can create an i-sound analysis by combining only i-sound components (all of our analyses have i-sound variations) in the pipeline. It is also possible to combine i-unsound components in the pipeline to trade precision for improved recall.

5.4 Dynamic Mutability Analysis

Our dynamic mutability analysis observes the program's execution and classifies as *mutable* those method parameters that are used to mutate objects. Our analysis does not implement the formal rules of Definition 1. It implements a simpler version of the formal rules that is designed for optimization.

The algorithm is m-sound: it classifies a parameter as *mutable* only when the parameter is mutated. The algorithm is also i-sound: it classifies all remaining parameters as *unknown*. Section 5.4.1 gives the idea behind the algorithm, and Section 5.4.2 describes an optimized implementation.

To improve the analysis results, we developed several heuristics (Section 5.4.3). Each heuristic carries a different risk of unsoundness. However, most are shown to be accurate in our experiments. The analysis has an iterative variation with random input generation (Section 5.4.4) that improves analysis precision and run-time.

5.4.1 Conceptual Algorithm

The conceptual algorithm is based on Definition 1. The algorithm maintains the mutability set for each reference during the program execution. The mutability set is the set of all formal parameters (from any method invocation on the call stack) whose fields were directly or indirectly accessed

to obtain the reference. When a reference `x` is side-effected (i.e., used in `x.f = y`), all formal parameters in `x`'s mutability set are classified as mutable. The algorithm implements the following set of data-flow rules based on the evaluation rules in Section 5.1.3.2.

1. On method entry, the algorithm adds each formal parameter (that is classified as *unknown*) to the parameter set of the corresponding actual parameter reference.

2. On method exit, the algorithm removes all parameters for the current invocation from the parameter sets of all references in the program.

3. Assignments, including pseudo-assignments for parameter passing and return values, propagate the parameter sets unchanged.

4. Field accesses also propagate the sets unchanged: the set of parameters for `x.f` is the same as that of `x`.

5. For a field write `x.f = v`, the algorithm classifies as *mutable* all parameters in the parameter set of `x`.

The next section presents an alternative algorithm that we implemented.

5.4.2 Optimized Dynamic Analysis Algorithm

Maintaining mutability sets for all references, as required by the algorithm of Section 5.4.1, is computationally expensive. To improve performance, we developed an alternative algorithm that does not maintain mutability sets. The alternative algorithm is i-sound and m-sound, but is less complete—it classifies fewer parameters. In the alternative algorithm, parameter p of method m is classified as *mutable* if: (i) the transitive state of the object that p points to changes during the execution of m, and (ii) p is not aliased to any other parameter of m. Without part (ii), the algorithm would not be m-sound—*immutable* parameters that are aliased to a *mutable* parameter during the execution might be wrongly classified as *mutable*.

The example code in Figure 5-10 illustrates the difference between the conceptual algorithm presented in Section 5.4.1 and the alternative algorithm presented in this section. By definition 1 parameters $p1, p3, p4$ are mutable. The conceptual algorithm will classify them correctly when observing the execution of the method `main`. However, the alternative algorithm leaves all parameters as *unknown* since these parameters are aliased (in fact in this example they refer to the same object)—when the modification occurs. Note that the intra-procedural static analysis (Section 5.5.1) compensates for the incompleteness of the dynamic analysis in this case and correctly classifies `p1, p3, p4` as *mutable*.

The algorithm permits an efficient implementation: when method m is called during the program's execution, the analysis computes the set $reach(m, p)$ of objects that are transitively reachable from each parameter p via field references. When the program writes to a field in object o, the analysis finds all parameters p of methods that are currently on the call stack. For each such parameter p, if $o \in reach(m, p)$ and p is not aliased to other parameters of m, then the analysis

```
1 class Main {
2     void m1(C p1, C p2) {
3       p1.x = null;
4     }
5     void m2(C p3, C p4) {
6       p3.x = null;
7       p4.x = null;
8     }
9
10    main(){
11        o = new C();
12        m1(o,o);
13        m2(o,o);
14    }
15 }
```

Figure 5-10: Example code that is used to illustrates the limitation of the alternative algorithm in Section 5.4.2.

classifies p as *mutable*. The algorithm checks aliasing by verifying emptiness of intersection of reachable sub-heaps (ignoring immutable objects, such as boxed primitives, which may be shared).

The implementation instruments the analyzed code at load time. The analysis works online, i.e., in tandem with the target program, without creating a trace file. Our implementation includes the following three optimizations, which together improve the run time by over 30×: (a) the analysis determines object reachability by maintaining and traversing its own data structure that mirrors the heap, which is faster than using reflection; (b) the analysis computes the set of reachable objects lazily, when a modification occurs; and (c) the analysis caches the set of objects transitively reachable from every object, invalidating it when one of the objects in the set is modified.

5.4.3 Dynamic Analysis Heuristics

The dynamic analysis algorithm described in Sections 5.4.1 and 5.4.2 is m-sound—a parameter is classified as *mutable* only if it is modified during execution. Heuristics can improve the completeness, or recall (see Section 5.6), of the algorithm. The heuristics take advantage of the *absence* of parameter modifications and of the classification results computed by previous stages in the analysis pipeline. Using the heuristics may potentially introduce i-unsoundness or m-unsoundness to the analysis results, but in practice, they cause few misclassifications (see Section 5.6.3.5).

(A) Classifying parameters as *immutable* at the end of the analysis. This heuristic classifies as *immutable* all (*unknown*) parameters that satisfy conditions that are set by the client of the analysis. In our framework, the heuristic classifies as *immutable* a parameter p declared in method m if p was not modified, m was executed at least N times, and the executions achieved block coverage of at least $t\%$. Higher values of the threshold N or t increase i-soundness but decrease completeness.

The intuition behind this heuristic is that, if a method executed multiple times, and the executions covered most of the method, and the parameter was not modified during any of those executions, then the parameter may be *immutable*. This heuristic is m-sound but i-unsound. In our experiments, this heuristic greatly improved recall and was not a significant source of mistakes (Section 5.6.3.5).

(B) Using current mutability classification. This heuristic classifies a parameter as *mutable* if the object to which the parameter points is passed in a method invocation to a formal parameter that is already classified as *mutable* (by a previous or the current analysis). That is, the heuristic does not wait for the actual modification of the object but assumes that the object will be modified if it is passed to a *mutable* position. The heuristic improves analysis performance by not tracking the object in the new method invocation.

The intuition behind this heuristic is that if an object is passed as an argument to a parameter that is known to be *mutable*, then it is likely that the object will be modified during the call. The heuristic is i-sound but m-unsound. In our experiments, this heuristic improved recall and run time of the analysis and caused few misclassifications (see Section 5.6.3.5).

(C) Classifying aliased mutated parameters. This heuristic classifies a parameter p as *mutable* if the object that p points to is modified, regardless of whether the modification happened through an alias to p or through the reference p itself. For example, if parameters a and b happen to point to the same object o, and o is modified, then this heuristic will classify both a and b as *mutable*, even if it the modification is only done using the formal parameter's reference to a.

The heuristic is i-sound but m-unsound. In our experiments, using this heuristic improved the results in terms of recall, without causing any misclassifications.

5.4.4 Using Randomly Generated Inputs

In this section we consider the use of randomly generated sequences of method calls as the required input for the dynamic analysis. Random generation can complement (or even replace) executions provided by a user. For instance, Pacheco et al. [111] uses feedback-directed random generation to detect previously-unknown errors in widely used (and tested) libraries.

Using randomly-generated execution has benefits for a dynamic analysis. First, the user need not provide a sample execution. Second, random executions may explore parts of the program that the user-supplied executions do not reach. Third, each of the generated random inputs may be executed immediately—this allows the client of the analysis to stop generating inputs when the client is satisfied with the results of the analysis computed so far. Fourth, the client of the analysis may focus the input generator on methods with unclassified parameters.

Our generator gives a higher selection probability to methods with *unknown* parameters and methods that have not yet been executed by other dynamic analyses in the pipeline. Generation of random inputs is iterative. After the dynamic analysis has classified some parameters, it makes sense to propagate that information (see Section 5.5.3) and to re-focus random input generation on the remaining *unknown* parameters. Such re-focusing iterations continue as long as each iteration classifies at least 1% of the remaining *unknown* parameters (the threshold is user-settable).

By default, we set the number of generated method calls per each iteration to be the maximum between 5000 and the number of methods in the program. The randomly generated inputs are

executed in a safe way [111], using a Java security manager.

5.5 Static Mutability Analysis

This section describes a simple, scalable static mutability analysis. It consists of two phases: S, an intraprocedural analysis that classifies as *(im)mutable* parameters (never) affected by field writes within the procedure itself (Section 5.5.2), and P, an interprocedural analysis that propagates mutability information between method parameters (Section 5.5.3). P may be executed at any point in an analysis pipeline after S has been run, and may be run multiple times, interleaving with other analyses. S and P both rely on an intraprocedural pointer analysis that calculates the parameters pointed to by each local variable (Section 5.5.1).

5.5.1 Intraprocedural Points-To Analysis

To determine which parameters can be pointed to by each expression, we use an intraprocedural, context-insensitive, flow-insensitive, 1-level field-sensitive, points-to analysis. As a special case, the analysis is flow-sensitive on the code from the beginning of a method through the first backwards jump target, which includes the entire body of methods without loops. We are not aware of previous work that has explored this point in the design space, which we found to be both scalable and sufficiently precise.

The points-to analysis calculates, for each local variable l, a set $P_0(l)$ of parameters whose state l can point to directly and a set $P(l)$ of parameters whose state l can point to directly or transitively. (Without loss of generality, we assume three-address SSA form and consider only local variables.) The points-to analysis has "overestimate" and "underestimate" varieties; they differ in how method calls are treated (see below).

For each local variable l and parameter p, the analysis calculates a distance map $D(l, p)$ from the fields of object l to a non-negative integer or ∞. $D(l, p)(f)$ represents the number of dereferences that can be applied to l starting with a dereference of the field f to find an object pointed to (possibly indirectly) by p. Each map $D(l, p)$ is either strictly positive everywhere or is zero everywhere. As an example, suppose l directly references p or some object transitively pointed to by p; then $D(l, p)(f) = 0$ for all f. As another example, suppose $l.f.g.h = p.x$; then $D(l, p)(f) = 3$. The distance map D makes the analysis field-sensitive, but only at the first layer of dereferencing; we found this to be important in practice to provide satisfactory results.

The points-to analysis computes $D(l, p)$ via a fixpoint computation on each method. At the beginning of the computation, $D(p, p)(f) = 0$, and $D(l, p)(f) = \infty$ for all $l \neq p$. The dataflow rules are straightforward, so we give their flavor with a few examples:

- A field dereference $l_1 = l_2.f$ updates

$$\begin{aligned} \forall g : D(l_1, p)(g) &\leftarrow \min(D(l_1, p)(g), D(l_2, p)(f) - 1) \\ D(l_2, p)(f) &\leftarrow \min(D(l_2, p)(f), \min_g D(l_1, p)(g) + 1) \end{aligned}$$

- A field assignment $l_1.f = l_2$ updates

$$\begin{aligned} D(l_1, p)(f) &\leftarrow \min(D(l_1, p)(f), \min_g D(l_2, p)(g) + 1) \\ \forall g : D(l_2, p)(g) &\leftarrow \min(D(l_2, p)(g), D(l_1, p)(f) - 1) \end{aligned}$$

- Method calls are handled either by assuming they create no aliasing (creating an underestimate of the true points-to sets) or by assuming they might alias all of their parameters together (creating an overestimate). If an underestimate is desired, no values of $D(l, p)(f)$ are updated. For an overestimate, let S be the set of all locals used in the statement (including receiver and return value); for each $l \in S$ and each parameter p, set $D(l, p)(f) \leftarrow \min_{l' \in S} \min_{f'} D(l', p)(f')$.

After the computation reaches a fixpoint, it sets

$$\begin{aligned} P(l) &= \{p \mid \exists f : D(l, p)(f) \neq \infty\} \\ P_0(l) &= \{p \mid \forall f : D(l, p)(f) = 0\} \end{aligned}$$

5.5.2 Intraprocedural Phase: S

The static analysis S works in four steps. First, S performs the "overestimate" points-to analysis (Section 5.5.1). Second, the analysis marks as *mutable* some parameters that are currently marked as *unknown*: for each mutation $l_1.f = l_2$, the analysis marks all elements of $P_0(l_1)$ as *mutable*. Third, the analysis computes a "leaked set" L of locals, consisting of all arguments (including receivers) in all method invocations and all locals assigned to a static field (in a statement of the form `Global.field = local`). Fourth, if all the parameters of a method that are not already classified as *immutable* are *unknown* parameters that are not in the set $\cup_{l \in L} P(l)$ the analysis marks them as *immutable*.

S is i-sound and m-unsound. To avoid over-conservatism, S assumes that on the entry to the analyzed method all parameters are fully un-aliased, i.e., point to disjoint parts of the heap. This assumption may cause S to miss possible mutations due to aliased parameters; to maintain i-soundness, S never classifies a parameter as *immutable* unless all other parameters to the method can be classified as *immutable*.

For example, S does not detect any mutation to parameter `p5` of the method `modifyAll` in Figure 5-1. Since other parameters of `modifyAll` (i.e., `p3` and `p4`) are classified as *mutable*, S conservatively leaves `p5` as *unknown*. In contrast, Sălcianu's static analysis JPPA [131] incorrectly classifies `p5` as *immutable*.

The m-unsoundness of S is due to infeasible paths (e.g., unreachable code), flow-insensitivity, and the overestimation of the points-to analysis.

5.5.2.1 Intraprocedural Analysis Heuristic

We have also implemented a i-unsound heuristic SH that is like S, but it can classify parameters as *immutable* even when other parameters of the same method are not classified as *immutable*. In our

experiments, this never caused a misclassification.

5.5.3 Interprocedural Propagation Phase: P

The interprocedural propagation phase P refines the current parameter classification by propagating both mutability and immutability information through the call graph. Given an i-sound input classification, the propagation algorithm is i-sound and m-sound. However, our implementation is i-sound and m-unsound.

Because propagation ignores the bodies of methods, the P phase is i-sound only if the method bodies have already been analyzed. It is intended to be run only after the S phase of Section 5.5.1 has already been run. However, it can be run multiple times (with other analyses in between).

Section 5.5.3.1 describes the binding multi-graph (BMG), and then Section 5.5.3.2 gives the propagation algorithm itself.

5.5.3.1 Binding Multi-Graph

The propagation uses a variant of the *binding multi-graph* (BMG) [37]; our extension accounts for pointer data structures. Each node is a method parameter `m.p`. An edge from `m1.p1` to `m2.p2` exists iff `m1` calls `m2`, passing as parameter `p2` part of `p1`'s state (either `p1` or an object that may be transitively pointed-to by `p1`).

A BMG is created by generating a call-graph and translating each method call edge into a set of parameter dependency edges, using the sets *P*(*l*) described in Section 5.5.1 to tell which parameters correspond to which locals.

The BMG creation algorithm is parameterized by a call-graph construction algorithm. Our experiments used CHA [46]—the simplest and least precise call-graph construction algorithm offered by Soot. In the future, we want to investigate using more precise but still scalable algorithms, such as RTA [15] (available in Soot, but containing bugs that prevented us from using it), or those proposed by Tip and Palsberg [141] (not implemented in Soot).

The true BMG is not computable, because determining perfect aliasing and call information is undecidable. Our analysis uses an under-approximation (i.e., it contains a subset of edges of the ideal graph) and an over-approximation (i.e., it contains a superset of edges of the ideal graph) to the BMG as safe approximations for determining mutable and immutable parameters, respectively. One choice for the over-approximated BMG is the *fully-aliased* BMG, which is created with an overestimating points-to analysis which assumes that method calls introduce aliasings between *all* parameters. One choice for the under-approximated BMG is the *un-aliased* BMG, which is created with an underestimating points-to analysis which assumes that method calls introduce *no* aliasings between parameters. More precise approximations could be computed by a more complex points-to analysis.

To construct the under-approximation of the true BMG, propagation needs a call-graph that is an under-approximation of the real call-graph. However, most existing call-graph construction algorithms [15, 46, 50, 141] create an over-approximation. Therefore, our implementation uses the same call-graph for building the un- and fully-aliased BMGs. Due to this approximation, our implementation of P is m-unsound. Actually, P is m-unsound even on the under-approximation

of the BMG. For example, assume that m1.p1 is unknown, m2.p2 is mutable, and there is an edge between m1.p1 and m2.p2. It is possible that there is an execution of m2.p2 in which p2 is mutated, but for every execution that goes through m1, m2.p2 is immutable. In this case, the algorithm would incorrectly classify m1.p1 as mutable. In our experiments, this approximation caused several misclassifications of *immutable* parameters as *mutable* (see Section 5.6.3.1).

5.5.3.2 Propagation Algorithm

Propagation refines the parameter classification in 2 phases.

The **mutability propagation** classifies as *mutable* all the *unknown* parameters that can reach in the under-approximated BMG (that is, can flow to in the program) a parameter that is classified as *mutable*. Using an over-approximation to the BMG would be unsound because spurious edges may lead propagation to incorrectly classify parameters as mutable.

The **immutability propagation** phase classifies additional parameters as *immutable*. This phase uses a fix-point computation: in each step, the analysis classifies as *immutable* all *unknown* parameters that have no *mutable* or *unknown* successors (callees) in the over-approximated BMG. Using an under-approximation to the BMG would be unsound because if an edge is missing in the BMG, the analysis may classify a parameter as *immutable* even though the parameter is really mutable. This is because the parameter may be missing, in the BMG, a *mutable* successor.

5.6 Evaluation

We implemented the combined static and dynamic analysis framework in a tool, Pidasa, and experimentally evaluated all sensible combinations (192 in all) of the mutability analyses described above, we compared the results with each other and with the correct classification of parameters as determined by Definition 1. Our results indicate that staged mutability analysis can be accurate, achieve better run-time performance, and are useful.

5.6.1 Methodology and Measurements

We computed mutability for 6 open-source subject programs (see Figure 5-11). When an example input was needed (e.g., for a dynamic analysis), we ran each subject program on a single input.

- **jolden**[1] is a benchmark suite of 10 small programs. As the example input, we used the `main` method and arguments that were included with the benchmarks. We included these programs primarily to permit comparison with Sălcianu's evaluation [131].
- **sat4j**[2] is a SAT solver. We used a file with an unsatisfiable formula as the example input.
- **tinysql**[3] is a minimal SQL engine. We used the program's test suite as the example input.

[1] `http://www-ali.cs.umass.edu/DaCapo/benchmarks.html`

[2] `http://www.sat4j.org/`

[3] `http://sourceforge.net/projects/tinysql`

program	size (LOC)	classes	parameters all	parameters non-trivial	parameters inspected
jolden	6,215	56	705	470	470
sat4j	15,081	122	1,499	1,136	118
tinysql	32,149	119	2,408	1,708	206
htmlparser	64,019	158	2,270	1,738	82
ejc	107,371	320	9,641	7,936	7,936
daikon	185,267	842	16,781	13,319	73
Total	410,102	1,617	33,304	26,307	8,885

Figure 5-11: Subject programs.

- **htmlparser**[4] is a real-time parser for HTML. We used our research group's webpage as the example input.
- **ejc**[5] is the Eclipse Java compiler. We used one Java file as the example input.
- **daikon**[6] is an invariant detector. We used the StackAr test case from its distribution as the example input.

As the input to the first analysis in the pipeline, we used a pre-computed classification for all parameters in the Java standard libraries. Callbacks from the library code to the client code (e.g., `toString()`, `hashCode()`) were analyzed under the closed world assumption in which all of the subject programs were included. The pre-computed classification was created once, and reused many times in all the experiments. A benefit of using this classification is that it covers otherwise un-analyzable code, such as native calls.

We measured the results only for non-trivial parameters declared in the application. That is, we did not count parameters with a primitive, boxed primitive, or `String` type, nor parameters declared in external or JDK libraries.

To measure the accuracy of each mutability analysis, we determined the correct classification (*mutable* or *immutable*) for 8,885 parameters: all of jolden and ejc, and 4 randomly-selected classes from each of the other programs. To find the correct classification, we first ran every tool available to us (including our analysis pipelines, Sălcianu's tool, and the Javarifier [119, 145] type inference tool for Javari). Then, we manually verified the correct classification for every parameter where any two tool results differed, or where only one tool completed successfully. In addition, we verified an additional 200 parameters, chosen at random, where all tools agreed. We found no instances where the tools agreed on the mutability result, but the result was incorrect.

Figure 5-12 and the tables in Section 5.6.3 present precision and recall results, computed as

[4] `http://htmlparser.sourceforge.net/`

[5] `http://www.eclipse.org/`

[6] `http://pag.csail.mit.edu/daikon/`

follows:

$$\begin{aligned}
\text{i-precision} &= \frac{ii}{ii+im} \\
\text{i-recall} &= \frac{ii}{ii+ui+mi} \\
\text{m-precision} &= \frac{mm}{mm+mi} \\
\text{m-recall} &= \frac{mm}{mm+um+im}
\end{aligned}$$

where *ii* is the number of immutable parameters that are correctly classified, and *mi* is the number of immutable parameters incorrectly classified as *mutable* (similarly, *ui*). Similarly, for mutable parameters, we have *mm*, *im* and *um*. i-precision is measure of soundness: it counts how often the analysis is correct when it classifies a parameter as *immutable*. i-recall is measure of completeness: it counts how many immutable parameters are marked as such by the analysis. m-precision and m-recall are similarly defined. An i-sound analysis has i-precision of 1.0, and an m-sound analysis has m-precision of 1.0. Ideally, both precision and recall should be 1.0, but this is not feasible: there is always a trade-off between analysis precision and recall.

5.6.2 Evaluated Analyses

Our experiments evaluate pipelines composed of analyses described in Section 5.3. X-Y-Z denotes a staged analysis in which component analysis X is followed by component analysis Y and then by component analysis Z.

Our experiments use the following component analyses:

- S is the sound intraprocedural static analysis (Section 5.5.2).
- SH is the intraprocedural static analysis heuristic (Section 5.5.2.1).
- P is the interprocedural static propagation (Section 5.5.3).
- D is the dynamic analysis (Section 5.4), using the inputs of Section 5.6.1.
- DH is D, augmented with all the heuristics described in Section 5.4.3. DA, DB, and DC are D, augmented with just one of the heuristics.
- DRH is DH enhanced with random input generation (Section 5.4.4); likewise for DRA, etc.

Additionally, we used JPPA [131]. Its informal mutability definition matches Definition 1, so JPPA is a natural comparison for the above analyses. We included the following additional analyses:

- J is Sălcianu and Rinard's state-of-the-art static analysis JPPA that never classifies parameters as *mutable*—only *immutable* and *unknown*.
- JM is J, augmented to use a `main` method that contains calls to all the public methods in the subject program [124]; J only analyzes methods that are reachable from `main`.
- JMH is JM plus an m-unsound heuristic to classify as *mutable* any parameter for which J provides an explanation of a potential modification.

5.6.3 Results

We experimented with six programs and 192 different analysis pipelines. Figure 5-12 compares the accuracy of a selected set of mutability analyses among those with which we experimented. S-P-DRBC-P is the best-performing i-sound staged analysis. For clients that do not require i-soundness, the pipeline with the highest sum of precision and recall was SH-P-DRH-P. Compared to Sălcianu's [131] state-of-the-art analysis J, the staged mutability analysis achieves equal or slightly worse i-precision, better i-recall, and much better m-recall and m-precision. SH-P-DRH-P also achieves better run-time performance (see section 5.6.4). Sălcianu's analysis augmented with a heuristic for classifying mutable references, followed by our best stage analysis, JMH-SH-P-DRH-P, achieves the highest i-recall.

This section discusses the important observations that stem from the results of our experiments. Each sub-section discusses one observation that is supported by a table listing representative pipelines illustrating the observation. The tables in this section present results for ejc. Results for other programs were similar. However, for smaller programs all analyses did better and the differences in results were not as pronounced.

5.6.3.1 Interprocedural Propagation

Running interprocedural propagation (P in the tables) is always beneficial, as the following table shows on representative pipelines.

Analysis	i-recall	i-precision	m-recall	m-precision
SH	0.563	1.000	0.299	0.998
SH-P	0.777	1.000	0.904	0.971
SH-P-DRH	0.922	0.996	0.906	0.971
SH-P-DRH-P	0.928	0.996	0.907	0.971
DRH	0.540	0.715	0.144	0.987
DRH-P	0.940	0.776	0.663	0.988

Propagation may decrease m-precision but, in our experiments, the decrease was never larger than 0.03 (not shown in the above table). In the experiments, propagation always increased all other statistics (sometimes significantly). For example, the table shows that propagation increased i-recall from 0.563 in SH to 0.777 in SH-P and it increased m-recall from 0.299 in SH to 0.904 in SH-P. Moreover, since almost all of the run-time cost of propagation lies in the call-graph construction, only the first execution incurs notable run-time cost on the analysis pipeline; subsequent executions of propagation are fast. Therefore, most pipelines presented in the sequel have P stages executed after each other analysis stage.

5.6.3.2 Combining Static and Dynamic Analysis

Combining static and dynamic analysis in either order is helpful—the two types of analysis are complementary.

Prog.	Analysis	i-recall	i-precision	m-recall	m-precision
ejc	S-P-DRBC-P	0.781	1.000	0.915	0.956
	SH-P	0.777	1.000	0.904	0.971
	SH-P-DRH-P	0.928	0.996	0.907	0.971
	J	0.593	0.999	0.000	0.000
	JMH	0.734	0.998	0.691	0.941
	JMH-SH-P-DRH-P	0.939	0.997	0.944	0.951
jolden	S-P-DRBC-P	0.829	1.000	1.000	0.924
	SH-P	0.829	1.000	0.907	1.000
	SH-P-DRH-P	0.973	1.000	1.000	0.970
	J	0.894	1.000	0.000	0.000
	JMH	0.985	1.000	0.660	0.955
	JMH-SH-P-DRH-P	0.989	0.996	0.990	0.970
daikon	S-P-DRBC-P	0.705	1.000	0.931	0.844
	SH-P	0.636	1.000	0.931	0.844
	SH-P-DRH-P	0.750	1.000	0.931	0.844
	J	0.750	1.000	0.000	0.000
	JMH	-	-	-	-
	JMH-SH-P-DRH-P	-	-	-	-
tinysql	S-P-DRBC-P	0.965	1.000	0.655	0.655
	SH-P	0.955	1.000	0.667	0.800
	SH-P-DRH-P	0.980	0.995	0.667	0.667
	J	-	-	-	-
	JMH	-	-	-	-
	JMH-SH-P-DRH-P	-	-	-	-
sat4j	S-P-DRBC-P	0.763	1.000	0.968	0.968
	SH-P	0.763	1.000	0.968	0.968
	SH-P-DRH-P	0.854	0.980	0.968	0.968
	J	-	-	-	-
	JMH	-	-	-	-
	JMH-SH-P-DRH-P	-	-	-	-
htmlparser	S-P-DRBC-P	0.482	1.000	0.962	1.000
	SH-P	0.482	1.000	0.654	1.000
	SH-P-DRH-P	0.978	0.989	0.962	0.976
	J	-	-	-	-
	JMH	-	-	-	-
	JMH-SH-P-DRH-P	-	-	-	-

Figure 5-12: Mutability analyses on subject programs. Empty cells mean that the analysis aborted with an error. The abbreviations for the component analysis are described in Section 5.6.2.

Analysis	i-recall	i-precision	m-recall	m-precision
SH-P	0.777	1.000	0.904	0.971
SH-P-DRH	0.922	0.996	0.906	0.971
SH-P-DRH-SH-P	0.928	0.996	0.907	0.971
DRH	0.540	0.715	0.144	0.987
DRH-SH-P	0.939	0.812	0.722	0.981
DRH-SH-P-DRH	0.943	0.813	0.722	0.981

For best results, the static stage should precede the dynamic stage. Pipeline SH-P-DRH, in which the static stage precedes the dynamic stage, achieved better i-precision and m-recall than DRH-SH-P, with marginally lower (by 0.01–0.02) i-recall and m-precision.

Repeating executions of static or dynamic analyses bring no substantial further improvement. For example, SH-P-DRH-SH-P (i.e., static-dynamic-static) achieves essentially the same results as SH-P-DRH (i.e., static-dynamic). Similarly, DRH-SH-P-DRH (i.e., dynamic-static-dynamic) only marginally improves i-recall over DRH-SH-P (i.e., dynamic-static).

5.6.3.3 Comparing Static Stages

In a staged mutability analysis, using a more complex static analysis brings little benefit. We experimented with replacing our lightweight interprocedural static analysis with J, Sălcianu's heavyweight static analysis.

Analysis	i-recall	i-precision	m-recall	m-precision
SH-P-DRH-P	0.928	0.996	0.907	0.971
J-DRH-P	0.973	0.787	0.664	0.998
JMH-DRH-P	0.939	0.922	0.878	0.949
JMH-SH-P-DRH-P	0.939	0.997	0.944	0.951

SH-P-DRH-P outperforms JMH-DRH-P with respect to 3 of 4 statistics, including i-precision (see Section 5.6.3.6). Combining the two static analyses improves recall—JMH-SH-P-DRH-P has better i-recall than SH-P-DRH-P and better m-recall than JMH-DRH-P. This shows that the two kinds of static analysis are complementary.

5.6.3.4 Randomly Generated Inputs in Dynamic Analysis

Using randomly generated inputs to the dynamic analysis (DRH) achieves better results than using a user-supplied execution (DH), at least for the relatively small user-supplied inputs (described in Section 5.6.1) with which we experimented. Although random generation can outperform or improve the results of a single user supplied input, Future work should evaluated whether random input generation can outperform and or augment the use of an exhaustive test suite.

Analysis	i-recall	i-precision	m-recall	m-precision
SH-P-DH	0.827	0.984	0.911	0.961
SH-P-DH-P-DRH	0.917	0.984	0.915	0.958
SH-P-DRH	0.922	0.996	0.906	0.971
SH-P-DRH-P-DH	0.932	0.983	0.912	0.970

Pipeline SH-P-DRH achieves better results than SH-P-DH with respect to i-precision, i-recall and m-precision (with lower m-recall). Using both kinds of executions can have different effects. For instance, SH-P-DH-P-DRH has better results than SH-P-DH, but SH-P-DRH-P-DH has a lower i-precision (due to i-unsoundness of heuristic **A**) with a small gain in i-recall and m-recall over SH-P-DRH.

The surprising finding that randomly generated code is as effective as using an example execution suggests that other dynamic analyses (e.g., race detection [107, 132], invariant detection [59], inference of abstract types [71], and heap type inference [116]) might also benefit from replacing example executions with random executions.

5.6.3.5 Dynamic Analysis Heuristics

By exhaustive evaluation, we determined that each of the heuristics is beneficial. A pipeline with DRH achieves notably higher i-recall and only slightly lower i-precision than a pipeline with DR (which uses no heuristics). This section indicates the unique contribution of each heuristic, by removing it from the full set (because some heuristics may have overlapping benefits). For consistency with other tables in this section, we present the results for ejc; however, the effects of heuristics were more pronounced on other benchmarks.

Heuristic **A** (evaluated by the DRBC line) has the greatest effect; removing this heuristic lowers i-recall (as compared to SH-P-DRH-P, which includes all heuristics.) However, because the heuristic is i-unsound, using it decreases i-precision, albeit only by 0.004. Heuristic **B** (the DRAC line) increases both i-recall and i-precision, and improves performance by 10%. Heuristic **C** (the DRAB line) is primarily a performance optimization. Including this heuristic results in a 30% performance improvement and a small increase in m-recall.

Analysis	i-recall	i-precision	m-recall	m-precision
SH-P-DR-P	0.777	1.000	0.905	0.971
SH-P-DRH-P	0.928	0.996	0.907	0.971
SH-P-DRBC-P	0.777	1.000	0.906	0.971
SH-P-DRAC-P	0.927	0.995	0.905	0.971
SH-P-DRAB-P	0.928	0.996	0.906	0.971

Heuristic **A** is parameterized by a coverage threshold t. Higher values of the threshold classify fewer parameters as *immutable*, increasing i-precision but decreasing i-recall. Figure 5-13 shows this relation. The heuristic is m-sound, so it has no effect on m-precision. The threshold value may reduce m-recall (if the analysis incorrectly classifies a *mutable* parameter), but, in our experiments, we have not observed this.

5.6.3.6 i-sound Analysis Pipelines

An i-sound mutability analysis never incorrectly classifies a parameter as *immutable*. All our component analyses have i-sound variations, and composing i-sound analyses yields an i-sound staged analysis.

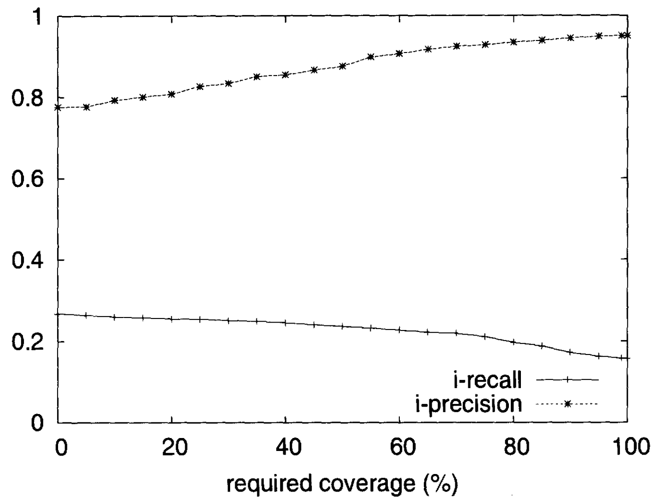


Figure 5-13: Relation between i-precision, i-recall, and the coverage threshold in dynamic analysis heuristic A. The presented results are for the dynamic analysis DA on the ejc subject program.

Analysis	i-recall	i-precision	m-recall	m-precision
S	0.454	1.000	0.299	0.998
S-P	0.777	1.000	0.904	0.971
S-P-DRBC-P	0.777	1.000	0.906	0.971
S-P-DBC-P	0.777	1.000	0.912	0.959

S is the i-sound intra-procedural static analysis. Not surprisingly, the i-sound pipelines achieve lower i-recall than the i-unsound pipelines presented in Figure 5-12; (Figure 5-12 also presents the i-sound results for S-P-DRBC-P for all subjects.) For clients for whom i-soundness is critical, this may be an acceptable trade-off. In contrast to our analyses, J is not i-sound [130], although it did achieve very high i-precision (see Figure 5-12).

5.6.4 Run-time Performance

Figure 5-14 shows run times of analyses on daikon (185 kLOC, which is considerably larger than subject programs used in previous evaluations of mutability analyses [124, 126, 131]). The experiments were run using a quad-core AMD Opteron 64-bit 4×1.8GHz machine with 4GB of RAM, running Debian Linux and Sun HotSpot 64-bit Server VM 1.5.0_09-b01. Experiments were run on single thread and used one core. Staged mutability analysis achieves better run-time performance on large code-bases and runs in about a quarter the time of Sălcianu's analysis (J in Figure 5-14).

Figure 5-14 shows that S-P (Section 5.6.3.6) runs, on daikon, an order of magnitude faster than J (or even better, if differences in call graph construction are discounted). Moreover, S-P is i-sound, while J is i-unsound. Finally, S-P has high m-recall and m-precision, while J has 0

Analysis	total time	last component
S	167	167
S-P	564	397
SH	167	167
SH-P	564	397
SH-P-DH	859	295
SH-P-DH-P	869	10
SH-P-DRH	1484	920
SH-P-DRH-P	1493	9
J	5586	5586
JM	-	-
JMH	-	-

Figure 5-14: The cumulative run time, and the time for the last component analysis in the pipeline, when analyzing the daikon subject program. Time is in seconds. Empty cells indicate that the analysis aborted with an error.

m-recall and m-precision.

An optimized implementation of the P and DRH stages could run even faster. First, the major cost of propagation (P) is computing the call graph, which could be reused later in the same pipeline. J's RTA [15] call graph construction algorithm takes seconds, but our tool uses Soot, which takes two orders of magnitude longer to perform CHA [46] (a less precise algorithm). Use of a more optimized implementation could greatly reduce the cost of propagation. Second, the DRH step iterates many times, each time performing load-time instrumentation and other tasks that could be cached; without this repeated work, DRH can be much faster than DH . We estimate that these optimizations would save between 50% and 70% of the total SH-P-DRH-P time.

There is a respect in which our implementation is more optimized than J. J is a whole-program analysis that cannot take advantage of pre-computed mutability information for a library such as the JDK. By contrast, our analysis does so by default, and Figure 5-14's numbers measure executions that use this pre-computed library mutability information. The number of annotated library methods is less than 10% of the number of methods in daikon.

5.6.5 Application: Test Input Generation

Section 5.6.3 evaluated the accuracy of mutability analyses. This section evaluates their utility by measuring how much the computed immutability information helps a client analysis. The client analysis is Palulu which was presented in chapter 2, a tool for generating regression tests.

Palulu combines dynamic analysis with random testing to create legal test inputs, where each input is a sequence of method calls. Palulu works even for programs in which most random sequences of method calls are illegal, and it does not require a formal specification.

Palulu operates in two steps. First, Palulu infers a model that summarizes the sequences of method calls (and their input arguments) observed during the example execution. Palulu generates a model for each class. The model is a directed graph where each edge corresponds to a method

analysis	nodes	ratio	edges	ratio	time (s)	ratio
jolden + ejc + daikon						
no immutability	444,729	1.00	624,767	1.00	6,703	1.00
SH-P-DRH-P	124,601	3.57	201,327	3.10	4,271	1.56
J	131,425	3.83	210,354	2.97	4,626	1.44
htmlparser + tinysql + sat4j						
no immutability	48,529	1.00	68,402	1.00	215	1.00
SH-P-DRH-P	8,254	5.88	13,047	5.24	90	2.38
J	-	-	-	-	-	-

Figure 5-15: Palulu (Chapter 2) model size and model generation time, when assisted by immutability classifications. The numbers are sums over indicated subject programs. Models with fewer nodes and edges are better. Also shown are improvement ratios over no immutability information (the "ratio" columns); larger ratios are better. Empty cells indicate that the analysis aborted with an error.

call, and each node corresponds to an abstract state of an observed instance. The model contains every sequence of method calls that occurred in the example execution.

Second, Palulu uses the inferred models to guide a feedback-directed random input generator [111] in creating legal and behaviorally-diverse method sequences. The test generator uses the model to decide which method should be called and what constants can be used as parameters.

An intermediate method in the sequence of method calls (that is, not the final method call about which some property is asserted) is useful only if it produces a new abstract value that can be used later in the sequence. The new abstract value can come from the method's return value or from side-effecting some existing value. In the absence of other information, Palulu assumes that every method can side-effect every parameter. For example, it would generate both of these method sequences:

```
Date d = new Date();
assert d.getTime() >= 0;
```

```
Date d = new Date();
boolean b = d.equals(null);
assert d.getTime() >= 0;
```

because the `equals` method might side-effect its receiver.

The model can be pruned (without changing the state space it describes) by removing calls that do not mutate specific parameters, because non-mutating calls are not useful in constructing behaviorally-diverse test inputs. A smaller model permits a systematic test generator to explore the state space more quickly, or a random test generator to explore more of the state space. Per-parameter mutability information permits more model reduction than method-level purity information.

We ran Palulu on our subject programs using no immutability information, and using immutability information computed by J and by SH-P-DRH-P. Since exhaustive model exploration is generally infeasible, Palulu can use unsound immutability information to improve its likelihood of finding errors within a given time bound.

Using the mutability information, Palulu's first step generated smaller models. Using the smaller models, Palulu's second step generated tests that achieved better line and block coverage

program	size (LOC)	classes	parameters all ⊇	non-trivial ⊇	inspected ⊇	immutable
jolden	6,215	56	705	470	470	277
tinysql	32,149	119	2,408	1,708	206	200
htmlparser	64,019	158	2,270	1,738	82	56
ejc	107,371	320	9,641	7,936	7,936	3,679
Total	209,754	653	15,033	11,852	8,994	4,212

Figure 5-16: Subject program characteristics. The **all** column lists the total number of parameters in the subject program. **non-trivial** parameters are not of primitive type or known immutable type (`java.lang.String` and boxed primitive types from package `java.lang`). Trivial parameters are always immutable. The **inspected** column lists the number of non-trivial parameters that we manually classified in each subject program (Section 5.6.1 describes how the parameters were selected for classification). Finally, the **immutable** column lists the number of inspected parameters that we manually determined to be immutable.

of the subject programs. Figure 5-15 shows the number of nodes and edges in the generated model graph, and the time Palulu took to generate the model (not counting the immutability analysis). Our experiment used J rather than JM, in part because JM runs on fewer programs, but primarily because JM's results would be no better than J . Palulu models include only methods called during execution, and J starts from the same main method. JM adds more analysis contexts, but doing so never changes a mutable parameter to immutable, which is the only way to improve Palulu model size.

5.7 Tool Comparison

Definition 1 is useful for evaluating both annotation-based systems expressing reference immutability and inference-based systems inferring reference immutability. We experimentally evaluated four tools that each infer an approximation to a different mutability definition, by comparing the tool results to Definition 1. We used the Pidasa and JMH tools described in Section 5.6, as well as Javari [19, 146] and JQual [69], both of which are static inference tools.

Our results indicate that the evaluated tools produce results that are similar to the formal definition we propose. This finding supports the practical utility of a common formal definition. Our results also highlight the differences between the expressiveness of the evaluated tools, i.e., which immutable references each tool identifies.

Section 5.7.1 summarizes the four tools we compared against Definition 1. Section 5.7.2 *quantitatively* compares the formal definition of reference immutability to each of the four evaluated tools. Section 5.7.3 presents *qualitative* comparisons that illustrate the major conceptual differences between our definition and the definitions implemented by the four tools.

5.7.1 Tools compared

We evaluated four tools:

- Pidasa is the tool that implements the static and dynamic analysis described in Section 5.6. We used the two best combinations: S-P-DRBC-P and SH-P-DRH-P. The tool for combination S-P-DRBC-P, denoted Pidasa_{snd}, is i-sound. The tool for combination SH-P-DRH-P, denoted Pidasa_{uns}, is i-unsound, but achieves better recall. Pidasa may also leave parameters unclassified; in the evaluation, we conservatively treated such parameters as mutable.

- JMH is a static analysis tool for computing reference immutability [131], described in Section 5.6.2.

- Javari is a type-annotation-based Java extension for specifying and enforcing reference immutability [19, 146]. The Javarifier tool [38, 119] inferred Javari annotations for the subject programs. To match the assumptions of the other tools, we ran Javarifier with the assumption that all fields should be considered part of the abstract state of an object, even though Javari and Javarifier support excluding specific fields from the abstract state of an object. Then, we conservatively changed all inferred `@Polyread` annotations (parametric polymorphism over mutability) to mutable (15 for jolden, 39 for htmlparser, 70 for tinysql, and 112 for eclipsec). We did not change the Javarifier output in any other way. The resulting annotated programs typecheck in accordance to the Javari definition.

- JQual is a static analysis tool for computing reference immutability [69]. We used the version downloaded from `http://www.cs.umd.edu/projects/PL/jqual` (10 December, 2007). We ran JQual in the context-insensitive, field-insensitive mode, because in any other mode it is unscalable [69] and could process none of the subject programs. Thus, JQual did not infer its parametric polymorphism over mutability.

Regrettably, we did not find more programs on which all of the tools run to completion—bugs in the tools make them terminate with errors.

5.7.2 Quantitative Comparison

Figure 5-17 presents the results of the quantitative analysis. In this section, we take precision and recall to mean i-precision and i-recall, respectively. Precision and recall of 1.0 would indicate perfect agreement with the definition. This evaluation uses precision and recall *not* as a measure of analysis quality (i.e., how well an analysis classifies parameters according to its own definition), but rather as a measure of how the mutability results of existing tools match our formal definition.

All four evaluated analyses are flow-insensitive, so none has perfect recall. Pidasa_{uns} achieves highest recall by sacrificing some precision. Javari and Pidasa_{snd} aim for perfect precision and accept low recall. JMH also has nearly perfect precision (the imprecision is due to an implementation bug), but the conservative static analysis may lead to low recall (e.g., in the largest subject program, ejc). JQual also has nearly perfect precision, and its low recall is due to a difference in its definition of field mutability, discussed in Section 5.7.3.4.

Prog	# params	analysis	i-recall	i-precision
jolden	470	$Pidasa_{uns}$	0.978	1.000
		$Pidasa_{snd}$	0.829	1.000
		JMH	0.985	1.000
		Javari	0.968	1.000
		JQual	0.696	0.965
tinysql	206	$Pidasa_{uns}$	0.980	0.995
		$Pidasa_{snd}$	0.965	1.000
		JMH	-	-
		Javari	0.956	1.000
		JQual	-	-
htmlparser	82	$Pidasa_{uns}$	0.978	0.989
		$Pidasa_{snd}$	0.482	1.000
		JMH	-	-
		Javari	0.756	1.000
		JQual	-	-
ejc	7936	$Pidasa_{uns}$	0.928	0.996
		$Pidasa_{snd}$	0.781	1.000
		JMH	0.734	0.998
		Javari	0.986	1.000
		JQual	-	-

Figure 5-17: Mutability analyses on subject programs. The **# params** columns list the numbers of parameters that we manually inspected for mutability (see Figure 5-16). Precision and recall are computed only for the set of inspected parameters (Section 5.7.2 describes the details). Empty cells mean that the analysis aborted with an error.

5.7.3 Qualitative Comparison

This Section presents a qualitative analysis describing more closely how each tool's results compare to Definition 1.

5.7.3.1 Pidasa

We examined all parameters on which Pidasa disagreed with the classification according to Definition 1. All differences can be attributed to imprecision in Pidasa's implementation. Thus, our examination showed that the definition of reference immutability used in Pidasa's technique, agrees with our formal definition.

$Pidasa_{uns}$ misclassified 23 mutable parameters as immutable due to an unsound heuristic in its dynamic component. The heuristic classifies a method parameter as immutable if no mutation occurred during the dynamic analysis phase and the block coverage of the method is above a certain threshold ($Pidasa_{uns}$ uses a 85% threshold).

$Pidasa_{uns}$ and $Pidasa_{snd}$ classified a parameter as `mutable` in several cases where the formal

definition classifies the parameter as `immutable`, because no mutation can occur at run time. The discrepancies are due to the following analysis imprecision:

- Flow-insensitivity of the static analysis component. Flow-insensitive analysis does not consider the order of statements in the method. This contrasts with Definition 1, which is flow-sensitive. Flow-insensitivity may lead Pidasa to classify some types to be mutable even though no mutation will ever occur at run-time.

  For example, in this code:

  ```
  void foo(Date d) {
    d = new Date();
    d.setHour(12);
  }
  ```

  a flow-insensitive analysis would classify parameter d as mutable because the variable d can be mutated, even though the value passed as the parameter cannot be mutated.

- Dynamic component failing to generate inputs that exercise a method. This results in the parameters being left unclassified, which, in this evaluation, we conservatively treated as `mutable`.

- Call-graph approximations in the static component. Because the precise caller-callee relationship cannot always be computed in an object-oriented language, the static component of Pidasa uses a conservative approximation of the call-graph.

- Heuristics in the dynamic component. For example, in the code

  ```
  void m(Object p1, Object p2){
    p1.field = ...;
  }
  ```

  if, at run-time, p1 and p2 happen to be aliased (i.e., there is an overlap between the parts of the heap reachable from p1 and p2), then the dynamic analysis heuristic incorrectly classifies p2 as mutable.

5.7.3.2 JMH

We examined all parameters on which the JMH results disagreed with the classification according to the formal Definition 1. All differences are due to either bugs, or imprecision in the inference tool. Thus, our examination showed that the (implicit) definition used by JMH agrees with our proposed formal definition of reference immutability.

In ejc, JMH misclassified 5 mutable parameters as *immutable*. More precisely, JMH classified as *immutable* 5 parameters that are *mutable* according to the proposed formal definition.

Program	Inspected parameters	Arrays	Flow insensitivity	Dynamic scope	Total differences
jolden	470	12	3	0	15
tinysql	206	9	0	0	9
htmlparser	82	6	12	2	20
ejc	7,936	17	1	31	49

Figure 5-18: Discrepancies where Javarifier classified a parameter reference as mutable, when it's classification by Definition 1 is immutable. For jolden, we manually analyzed all classes. For tinysql, htmlparser and ejc, we manually analyzed four classes selected at random. The columns Arrays, Flow Insensitive, and Dynamic indicate difference sources for the discrepancies as explained in Section 5.7.3.3.

This misclassification is due to what seems to be an incorrect treatment of the native method `System.arraycopy()`. `System.arraycopy()` copies objects from a source array to a destination array. JMH seems to treat the destination array as immutable, which is incorrect.

JMH fails to correctly classify immutable parameters (i.e., the computed result disagrees with the proposed formal definition, with JMH reporting *mutable* when the parameter cannot be mutated at run time) due to:

- Failure to analyze package-private constructors that are called in static field initializers.
- Failure to analyze non-abstract methods in abstract classes.
- Conservative pointer analysis. For example, the method `CompilationResult.computePriority()` in ejc calls the method `HashMap.get()` on a field. This receiver is classified as *mutable* by JMH, since `LinkedHashMap.get()` can mutate its receiver (optimized internal state). However, the specific field can only be instantiated with `HashMap` (super-class of `LinkedHashMap`) for which `get()` is a side-effect-free method. Since JMH is a closed-world analysis, in theory it could compute the correct results in this case.
- Call-graph approximations similar to Pidasa (Section 5.7.3.1).
- Flow-insensitivity of the analysis similar to Pidasa (Section 5.7.3.1).

5.7.3.3 Javari

The Javarifier type inference tool infers the reference mutability of every parameter according to the Javari mutability definition. Figure 5-18 tabulates the differences between the formal Definition 1 and Javari. Javari is perfectly i-precise: the only kind of difference is when Javarifier inferred a mutable type that Definition 1 classifies as an immutable type. Javarifier disagreed with Definition 1 on the mutability of a parameter for three reasons:

Arrays. In Javari immutability is not deep with respect to arrays (or generics)—a client can control the mutability of each level explicitly. Therefore, Javari allows different mutabilities for arrays and their elements, while our definition does not. When Javarifier inferred an immutable

```
1  class Client {
2    public static void main(String[] args) {
3      Client c = new Client();
4      Data d = new Data();
5      c.update(d);
6      c.mutateData();
7    }
8
9    private Data data;
10
11   // Formal definition: parameter is immutable
12   // Javari:            parameter is mutable
13   public void update(Data newData) {
14     data = newData;
15   }
16
17   public void mutateData() {
18     data.mutate();
19   }
20 }
```

Figure 5-19: The formal definition considers parameter newData to be immutable because update() does not itself mutate either data or newData. Javari considers newData to be mutable because update() stores newData for possible later mutation.

array of mutable elements for a parameter, for the purpose of comparison to Definition 1, we treat Javarifier's classification as a mutable parameter. Definition 1, however, classified the parameter as immutable.

Flow-insensitivity. Javari is a flow-insensitive type system (like most type systems, e.g., Java's), and thus can misclassify immutable parameters as mutable, similarly to Pidasa (Section 5.7.3.1).

Dynamic scope. The formal definition accounts for mutations that occur during the execution of a method. By contrast, Javari marks a parameter as mutable if the actual argument may later be mutated, even after the method has exited, as a result of being passed into the method. For example, consider Figure 5-19. As noted in Section 5.1.3.3, the formal definition can be modified to account for mutations even after a method has exited, simply by removing rule [R-Ret] of Figure 5-4. Future work could compare this modified formal definition to Javari.

Similarly, certain uses of parametric polymorphism force the Javari classification of some parameters to be mutable, while Definition 1 classifies them as immutable. Javari uses the @PolyRead type qualifier to express parametric polymorphism over mutability type qualifiers.[7] Definition 1

[7] A method with @PolyRead parameters can be viewed as having two signatures that are resolved via overloading:

```
1  class DateScanner {
2    private List<@ReadOnly Date> allDates;
3
4    // Formal definition: receiver is mutable, parameter is immutable
5    // Javari:            receiver is mutable, parameter is mutable
6    void addDate(Date date) {
7      Date scannedDate = scanDate(date);
8      allDates.add(scannedDate);
9    }
10
11   // Formal definition: receiver is immutable, parameter is immutable
12   // Javari:            receiver is polyread, parameter is polyread
13   // The polyread keyword after the parameter list annotates the
14   // receiver of the method as polyread.
15   void @PolyRead Date scanDate(@PolyRead Date newDate) @PolyRead {
16       ...
17     }
18   }
19 }
```

Figure 5-20: A program in which the parameter of addDate is passed into method scanDate, which takes a @Polyread parameter and receiver. Javari type rules require that parameter date of addDate be declared as mutable.

is unable to express this polymorphism. In Javari, @PolyRead parameters cannot be mutated explicitly in a method, so Definition 1 classifies these parameters as immutable.

For example, in Figure 5-20, the method addDate has a mutable receiver because it adds to the list allDates. Javari type-correctness requires the receiver types to be compatible when addDate calls scanDate. Since addDate's receiver is mutable, it uses the version of scanDate with a mutable receiver. But that version has a mutable parameter, and so date is marked as mutable, even though addDate does not mutate it (and no later mutation can occur when the Date is extracted from the list, because its type is @ReadonlyDate).

5.7.3.4 JQual

In the jolden subject program, JQual inferred 7 parameters to be readonly that Definition 1 states are mutable because it cannot be mutated at run-time.

in one version, all instances of @PolyRead are replaced by @Mutable, and in the other version, they are replaced by @ReadOnly. Almost every accessor method has its receiver and return type classified as @PolyRead. This means that when an accessor method is called on a @Mutable reference, the returned reference is @Mutable; when an accessor method is called on a @ReadOnly reference, the returned reference is @ReadOnly.

```
1  public class Counter {
2    int count;
3
4    public Counter head() {
5      return this;
6    }
7
8    // Formal definition: receiver mutable
9    // JQual:             receiver immutable
10   public void resetHead() {
11     head().reset();
12   }
13
14   public void reset() {
15     count = 0;
16   }
17 }
```

Figure 5-21: Example in which the proposed formal definition specifies that the receiver of resetHead() is mutable, while JQual classifies the receiver as immutable.

- In one case, JQual was more expressive when it inferred a parameter to be a readonly array of mutable objects, which the proposed formal definition classifies as a mutable array of mutable objects.
- In four cases, JQual incorrectly classified the receiver of a modifying method. Figure 5-21 demonstrates this problem when a mutating method is called on a reference that is passed through another accessor method.
- In 2 cases, JQual incorrectly classified the receiver of a method because the method was a member of an inner class.

In 84 other cases, JQual inferred a parameter to be mutable that is immutable according to Definition 1. We manually analyzed ten of these parameters, selected at random. All the differences were due to a definitional difference with regard to field mutability. In JQual, every method that reads a mutable field must have a mutable receiver, even if the method only reads the field and does not mutate the program state. This restriction makes JQual overly conservative in declaring mutable references, but helps ensure soundness in its type system. Figure 5-22 illustrates the problem.

In order to resolve this issue, JQual would need to run in a field-sensitive and context-sensitive mode. Running in a field-sensitive mode, JQual infers a separate type for each instance of a field, rather than a single static type for the field. Running in a context-sensitive mode, JQual can treat methods as polymorphic over mutability, similar to Javari's @Polyread. (Even though JQual

```
1  public class Info {
2    private int[] data;
3
4    // Formal definition: receiver immutable
5    // JQual:             receiver mutable
6    public int first() {
7      return data[0];
8    }
9
10   public void resetFirst() {
11     data[0] = 0;
12   }
13 }
```

Figure 5-22: Example in which the proposed formal definition specifies that the receiver of `first()` is reference immutable, while JQual classifies the receiver as mutable. Method `first()` reads `this.data`, so JQual requires that `this` have at least the same mutability as `data` (which is mutable).

cannot express this polymorphism over mutability in its final output, the method can be treated as polymorphic over mutability during the inference step.) In a field-sensitive and context-sensitive mode, JQual's inference is similar to Javari's inference. Specifically, in the class in Figure 5-22, the fact that the `data` field can be mutated in `resetFirst()` does not require reading the field in `first()` to be considered a mutation. As noted in Section 5.7.1, we were unable to run JQual in field-sensitive and context-sensitive mode, because JQual does not scale in this mode.

5.8 Related Work

Section 5.8.1 discusses previous work that discovers immutability (for example, determines when a parameter is never modified during execution). Section 5.8.2 discusses previous work that checks or enforces mutability annotations written by the programmer (or inserted by a tool).

5.8.1 Discovering Mutability

Early work [16,37] on analyzing programs to determine what mutations may occur considered only pointer-free languages, such as Fortran. In such a language, aliases are induced only by reference parameter passing, and aliases persist only until the procedure returns. Analyses that compute MOD set (modified parameters) determine which of the reference parameters, and which global variables, are assigned by the body of a procedure. Our static analysis extends this work to handle pointers and object-oriented programs, and incorporates field-sensitivity.

Subsequent research, often called side-effect analysis, addressed aliasing in languages containing pointers. An update `r.f` = v has the potential to modify any object that might be referred to by `r`. An alias analysis can determine the possible referents of pointers and thus the possible side effects. (An alias or class analysis also aids in call graph construction for object-oriented programs, by indicating the type of receivers and so disambiguating virtual calls.) This work indicates which aliased locations might also be mutated [87]—often reporting results in terms of the number of locations (typically, an allocation site in the program) that may be referenced—but less often indicates what other variables in the program might also refer to that site. More relevantly, it does not answer reference immutability questions regarding what references might be used to perform a mutation; ours is the first analysis to do so. A follow-on alias or escape analysis can be used to strengthen reference immutability into object immutability [19].

New alias/class analyses yield improved side-effect analyses [124, 128]. Landi *et al.* [88] improve the precision of previous work by using program-point-specific aliasing information. Ryder *et al.* [128] compare the flow-sensitive algorithm [88] with a flow-insensitive one that yields a single alias result that is valid throughout the program. The flow-sensitive version is more precise but slower and unscalable, and the flow-insensitive version provides adequate precision for certain applications. Milanova *et al.* [98] provide a yet more precise algorithm via an object-sensitive, flow-insensitive points-to analysis that analyzes a method separately for each of the objects on which the method is invoked. Object sensitivity outperforms Andersen's context-insensitive analysis [125]. Rountev [124] compares RTA to a context-sensitive points-to analysis for call graph construction, with the goal of improving side-effect analysis. Rountev's experimental results suggest that sophisticated pointer analysis may not be necessary to achieve good results. (This mirrors other work questioning the usefulness of highly complex pointer analysis [75, 127].) We, too, compared a sophisticated analysis (Sălcianu's) to a simpler one (ours) and found the simpler one competitive.

Side-effect analysis [30, 98, 124, 126, 130, 131] originated in the compiler community and has focused on i-sound analyses. Our work investigates other tradeoffs and other uses for the immutability information. Specifically, differently from previous research, our work (1) computes both *mutable* and *immutable* classifications, (2) trades off soundness and precision to improve overall accuracy, (3) combines dynamic and static stages, (4) includes a novel dynamic mutability analysis, and (5) permits an analysis to explicitly represent its incompleteness.

Preliminary results of using side effect analysis for optimization—an application that requires an i-sound analysis—show modest speedups. Le *et al.* [89] report speedups of 3–5% for a coarse CHA analysis, and only 1% more for a finer points-to analysis. Clausen [33] reports an average 4% speedup, using a CHA-like side effect analysis in which each field is marked as side-effected or not. Razafimahefa [120] reports an average 6% speedup for loop invariant code motion in an inlining JIT, Xu *et al.* [159] report slowdowns in a memoization optimization. Le *et al.* [89] summarize their own and related work as follows: "Although precision of the underlying analyses tends to have large effects on static counts of optimization opportunities, the effects on dynamic behavior are much smaller; even simple analyses provide most of the improvement."

Rountev [124] and Sălcianu [130, 131] developed static analyses for determining side-effect-free methods. Like our static analysis component, they combine a pointer analysis, an intra-

procedural analysis to determine "immediate" side effects, and inter-procedural propagation to determine transitive side effects. Sălcianu defines a side-effect-free method as one that does not modify any heap cell that existed when the method was called. Rountev's definition is more restricted and prohibits a side-effect-free method from creating and returning a new object, or creating and using a temporary object. Sălcianu's analysis can compute per-parameter mutability information in addition to per-method side effect information. (A method is side-effect-free if it modifies neither its parameters nor the global state, which is an implicit parameter.) Rountev's coarser analysis results are one reason that we cannot compare directly to his implementation. Rountev applies his analysis to program fragments by creating an artificial `main` routine that calls all methods of interest; we adopted this approach in augmenting J (see Section 5.6).

Sălcianu's [130, 131] analysis uses a complex pointer analysis. Its flow-insensitive method summary represents in a special way objects allocated by the current method invocation, so a side-effect-free method may perform side effects on a newly-allocated objects. Like ours, Sălcianu's analysis handles code that it does not have access to, such as native methods, by using manually prepared annotations. Sălcianu describes an algorithm for computing object immutability and proves it sound, but his implementation computes reference immutability (not object immutability) and contains some minor unsoundness. We evaluated our analyses, which also compute reference immutability, against Sălcianu's implementation (Section 5.5). In the experiments, our staged analyzed achieve comparable or better accuracy and better run-time performance.

Javarifier [38, 118, 119] infers the reference immutability type qualifiers of the Javari extension of Java [19, 146]. Starting at field reassignments (the source of all object side-effects), Javarifier flow- and context-sensitively propagates mutation information to all references, including method receivers. Our case studies using Javarifier (Section 5.7.3.3) show that Javari is sometimes more restrictive than our formal definition due to the conservative nature of its type rules. JQual [69] is a framework for inference of type qualifiers. JQual's definition of reference immutability and inference algorithm are similar to those of Javari.

Porat *et al.* [18, 117] infer class immutability for global (static) variables in Java's `rt.jar`, thus indicating the extent to which immutability can be found in practice; the work also addresses sealing/encapsulation. Foster *et al.* [61] developed an inference algorithm for `const` annotations using Cqual, a tool for adding type qualifiers to C programs. Their algorithm does not handle aliasing. Foster *et al.* also present a polymorphic version of `const` inference, in which a single reference may have zero or more annotations, depending on the context.

Other researchers have also explored the idea of dynamic side-effect analysis. Dallmeier and Zeller developed the JDynPur tool (`http://www.st.cs.uni-sb.de/models/jdynpur`) for offline dynamic side-effect analysis (not parameter mutability) but provide no description of the algorithm or experimental results. Xu *et al.* [159] developed dynamic analyses for detecting side-effect-free methods. Their work differs significantly from ours. Xu *et al.* consider only the method's receiver, while our analyses are more fine-grained and produce results for all formal parameters, including the receiver. Xu *et al.* examine only one analysis at a time. In contrast, our framework combines the strengths of static and dynamic analyses. Xu *et al.* do not present an evaluation of the effectiveness of their analyses in terms of precision and recall, they only report the percentage of methods identified as pure by their analyses. In contrast, we established the immutability of more than 8800

method parameters by manual inspection and report the results of our 192 analysis combinations with respect to the established ground truth. Finally, Xu *et al.*'s dynamic analysis is unsound. In contrast, our analysis framework is sound and we provide sound analyses, both static and dynamic, to use in the framework.

5.8.2 Specifying and Checking Mutability

A verification approach enforces reference immutability annotations written in the source code. Soundness requires that any cannot modify an object that was annotated as immutable. Since the problem is uncomputable, any static, sound system for checking annotations rejects some programs that cannot actually violate the immutability specifications at run-time.

Annotation-based approaches include Islands [77], Flexible Alias Protection [104], JAC [86], C++ `const` [138], ModeJava [134], Capabilities [22], Javari [19, 146], Universes [49], Relation types [148], and IGJ [164]. Many of these systems provide features beyond mutability annotations; for example, problems of ownership and aliasing can contribute to mutation errors, so a type system may also address those issues.

All the above approaches, except for Capabilities and C++'s `const` (which provide only non-transitive reference immutability), state as their goal the reference immutability described informally in the Introduction. Boyland [21] also noticed those similarities. In JAC [86], "the declarative definition of read-only types is: for an expression of a read-only type that evaluates to an object reference `r`, the full state of the referenced object is protected against changes performed via `r`." In ModeJava [134], "a read reference is a reference that is never used for modification of its referenced object (including retrieval of write references that may in turn be used for modification)." In Javari [19], "A read-only reference is a reference that cannot be used to modify the object to which it refers." In Universes [49], "references [...] must not be used to modify the referenced object since the reference is not guaranteed to come from the owner or a peer object of the modified object. Hence, we call these references readonly references."

However, while some of these descriptions are formal by having type rules, none specifies precisely what it means that a modification happens through the reference. Without a formal definition of reference immutability, it is not possible to compare different systems for inferring or checking it, nor is it possible to evaluate the systems' trade-offs between expressiveness and checkability. Capabilities [22], and Javari [19], do provide a formal definition for their systems, but do so using the programming language. In Javari (Capabilities are similar), a reference is read-only if it is possible to annotate it with readonly (i.e., the type system issues no error). In contrast, this chapter provides a formal definition that is independent of how it is calculated.

Object immutability is a stronger property than reference immutability: it guarantees that a particular value is never modified, even through aliased parameters. Reference immutability, together with an alias or escape analysis, is enough to establish object immutability [19]. Pechtchanski [113] allows the user to annotate his code with object immutability annotations and employs a combination of static and dynamic analysis to detect where those annotations are violated. The IGJ language [164] supports both reference and object immutability via a type system based on Java generics.

Side-effect analysis is different from parameter reference immutability, which is our focus. Side-effect analysis concerns methods and whether the heap can be modified during the method's execution. Parameter reference immutability concerns references to method parameters and whether they can be used to modify the state of objects. Except for very strict definitions of purity (such as strong purity [159]), method purity can often be computed from parameter reference immutability information (combined with analysis of globals).

5.9 Conclusion

We formally define parameter reference immutability. Previous work relied mostly on informal descriptions and type systems, both for inferring immutable references, and for checking annotations. The formal definition in this chapter encompasses those informal notions and enables unbiased comparisons between different inference and type-annotation approaches.

We have described Pidasa, a staged mutability analysis framework for Java, along with a set of component analyses that can be plugged into the analysis. The framework permits combinations of mutability analyses, including static and dynamic techniques. The framework explicitly represents analysis incompleteness and reports both immutable and mutable parameters. Our component analyses take advantage of this feature of the framework.

Our dynamic analysis is novel, to the best of our knowledge; at run time, it marks parameters as mutable based on mutations of objects. We presented a series of heuristics, optimizations, and enhancements that make it practical. For example, iterative random test input generation appears competitive with user-supplied sample executions. Our static analysis reports both *immutable* and *mutable* parameters, and it demonstrates that a simple, scalable analysis can perform at a par with much more heavyweight and sophisticated static analyses. Combining the lightweight static and dynamic analyses yields a combined analysis with many of the positive features of both, including both run-time performance and accuracy.

Our evaluation of Pidasa, includes many different combinations of staged analysis, in both sound and unsound varieties. This evaluation sheds insight into both the complexity of the problem and the sorts of analyses that can be effectively applied to it. We also show how the results of the mutability analysis can improve client analyses.

We compared parameter immutability in four systems (a type system Javari, and three analysis tools: JMH, Pidasa and JQual), on a large set of parameters in several programs. We then compared the results to the classification based on the formal definition, and analyzed the discrepancies. The results provide insight into the trade-offs each system makes between expressibility and verifiability. We observed that different systems vary in their approach to that trade-off. Javari is a type system and its type rules are conservative with respect to immutability, which leads Javari to mark some parameter references as mutable when in fact they cannot be used in a mutation. JMH is able to infer more immutable parameters than Javari, but it also is conservative and thus under-approximates the set of immutable references. Pidasa's inference combines static and dynamic analyses and offers both a conservative variant and a non-conservative variant that achieves the highest recall, with a small loss in precision.

Our case studies show that existing systems for expressing and inferring reference immutability

approximate the formal definition we present in this chapter. We hope that future research in this area finds the formal definition useful as a point of reference for verifying correctness and assessing expressibility.

Chapter 6

Conclusions

This chapter summarizes this dissertation and proposes several directions for future work.

This dissertation has demonstrated the how automatic test generation can be applied to address various issues of software bugs depending on the stake-holder involved with addressing the bugs. In particular we presented techniques for helping developers prevent new bugs in working functionality, helping testers detect existing bugs, and helping maintainers eliminate manifested bugs in deployed programs. We also show a mutability inference technique that increases the effectiveness of the other techniques.

- For developers we presented a technique for preventing new bugs in working programs targets object-oriented programs with constrained APIs. For these types of programs it is hard to generate legal instances and exercise method functionality without understanding the implicit constraints each method is making on its arguments.

  Our technique automatically generates structurally complex inputs in two phases. First, it creates models that describe the behavior of objects in given example executions. Second, it generates regression tests by exploring the space described by these models.

  Our tool Palulu implements our technique for Java. Palulu effectively generates unit-level regression tests with high coverage, and is capable of creating instances with highly complex structure that would require significant effort to create manually.

- For testers we presented a technique for detecting bugs in existing programs targets dynamic web applications, where bugs result from low testing standards, the dynamicity of the programming languages, and the effects of user interaction on executed functionality and output.

  Our technique creates test that expose failures in web-applications. In particular it attempts to find execution failures and malformed output. The technique iteratively executes the program while learning (and using) the relations between inputs and execution paths. Our technique simulates user input interactions, and checks the output using oracles. For each found failure, our technique localizes failures to code statements, and minimizes the inputs for each detected failure.

Our tool Apollo implements our technique for PHP and uses HTML validators as oracles. Apollo found 302 faults in 6 open-source PHP web applications. Out of these, 84 were execution problems and 218 malformed HTML.

- For maintainers we presented a technique that is helpful in eliminating bugs found in deployed object-oriented programs. Our technique captures partial snapshots of the state of method's arguments. When the program fails, the technique uses the captured state to create a test suite, where each test deterministically reproduces the failure.

  Our tool ReCrashJ implements our technique for Java. ReCrashJ is scalable, effective, and generates helpful test cases. ReCrashJ was able to reproduce real crashes from several programs with low overhead.

- Mutability inference allows a client analysis to avoid unnecessary work. Our prevention technique uses this information to prune the generated models. Our elimination technique uses this information to reduce the amount of information that needs to be copied and stored.

  We have formally defined parameter reference immutability, since previous work relied mostly on informal descriptions and type systems. The definition allows us to compare the immutability characteristics of inference tools and type-annotation approaches.

  We compared parameter immutability in four systems on a large set of parameters in several programs. The results provide insight into the trade-offs each system makes between expressibility and verifiability. Our results shows that most existing systems approximate our formal definition.

  Our technique infers the mutability of methods' arguments using a staged mutability analysis framework. The framework allows for a combination of component analyses, where each component in the pipeline explicitly represents its imprecision, and targets the imprecision of the previous analyses. We showed that the mutability inference analysis combining both static and dynamic analysis components in the staged analysis is scalable and accurate.

  Our tool Pidasa implements our technique for Java. Pidasa implements the framework, a static intra-procedural analysis, a static inter-procedural analysis, a novel dynamic analysis, as well as several heuristics for each analysis. We have tried different combinations of the components analyses and heuristics in Pidasa, with both sound and unsound varieties. Our evaluation shows that Pidasa allows for a very accurate and scalable solution.

6.1 Future Work

Enhancing Palulu models Methods in object-oriented programs expect arguments to be in a certain states in order to exercise functionality. Currently, Palulu models do not store any constraints on the state of methods' arguments, and rely on the generation phase to randomly use instances in different states as arguments to methods. A possible extension to Palulu will explore the relationship between the expressiveness of models with constraints on methods' arguments, and the practicality of exploring these models.

Effectiveness of Palulu technique We have evaluated Palulu on four subject programs. Palulu's effectiveness differed greatly between the subject programs. It would be interesting to understand the program characteristics for which our technique works best, or to improve its performance on the other subject programs.

Improving fault localization in Apollo Apollo localizes faults by combing the oracle output error positions with a mapping of the code statements to output. We can improve Apollo's fault localization by comparing executed statements in faulty and correct execution in a similar way to [83].

Client-side analysis in Apollo Apollo learns the connections between input and execution paths by analyzing scripts executed on the server, and HTML output. As a result, Apollo's coverage is limited on applications that process scripts on the client-side (e.g., using JavaScript). It would be valuable to adapt Apollo's technique to the challenges of analyzing client scripts, and combine it with the current server-side scripts analysis done in Apollo to improve coverage.

ReCrash state capture ReCrash is limited in reproducing failures that depend on concurrency, randomization, and external resources. ReCrash might be able to reproduce failures in cases where the failure source is incorrect handling of information that was generated (and stored) due to one of these reasons. We can increase ReCrash's effectiveness in reproducing this types of failures by storing relevant parts of the environment, and thread context switching information for methods accessing shared resources.

Fault localization using ReCrash ReCrash generates tests whenever a failure is detected. Thus, if ReCrash is used by multiple users the same failure may be exposed by several ReCrash-enabled applications. Thus, we can combine ReCrash with existing fault localization techniques (e.g., Tarantula [83]) that correlate code statements with faults by crossing coverage information between passing and failing tests.

Bibliography

[1] Apple Crash Reporter, 2007. http://developer.apple.com/technotes/tn2004/tn2123.html.

[2] Java Platform Debugger Architecture, 2007. http://java.sun.com/javase/technologies/core/toolsapis/jpda/.

[3] JVMTI Tool Interface (JVM TI), 2007. http://java.sun.com/j2se/1.5.0/docs/guide/jvmti/index.html.

[4] Microsoft Online Crash Analysis, 2007. http://oca.microsoft.com.

[5] Talkback Reports, 2007. http://talkback-public.mozilla.org.

[6] XStream Project Homepage, 2007. http://xstream.codehaus.org/.

[7] Glenn Ammons, Rastislav Bodík, and James R. Larus. Mining specifications. In *Proceedings of the 29th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 4–16, Portland, Oregon, January 16–18, 2002.

[8] Saswat Anand, Patrice Godefroid, and Nikolai Tillmann. Demand-driven compositional symbolic execution. In *TACAS*, 2008.

[9] Dana Angluin and Carl H. Smith. Inductive inference: Theory and methods. *ACM Computing Surveys*, 15(3):237–269, September 1983.

[10] Shay Artzi, Michael D. Ernst, Adam Kieżun, Carlos Pacheco, and Jeff H. Perkins. Finding the needles in the haystack: Generating legal test inputs for object-oriented programs. In *M-TOOS 2006: 1st Workshop on Model-Based Testing and Object-Oriented Systems*, Portland, OR, USA, October 23, 2006.

[11] Shay Artzi, Adam Kieżun, Julian Dolby, Frank Tip, Danny Dig, Amit Paradkar, and Michael D. Ernst. Finding bugs in dynamic web applications. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 261–272, Seattle, WA, USA, July 22–24, 2008.

[12] Shay Artzi, Adam Kieżun, David Glasser, and Michael D. Ernst. Combined static and dynamic mutability analysis. In *ASE 2007: Proceedings of the 22nd Annual International Conference on Automated Software Engineering*, pages 104–113, Atlanta, GA, USA, November 7–9, 2007.

[13] Shay Artzi, Sunghun Kim, and Michael D. Ernst. Recrash: Making software failures reproducible by preserving object states. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, pages 542–565, Paphos, Cyprus, July 9–11, 2008.

[14] Shay Artzi, Jaime Quinonez, Adam Kieżun, and Michael D. Ernst. A formal definition and evaluation of parameter immutability. *Automated Software Engineering*, 16(1):145–192, 2009.

[15] David F. Bacon and Peter F. Sweeney. Fast static analysis of C++ virtual function calls. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 324–341, San Jose, CA, USA, October 6–10, 1996.

[16] John P. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Proceedings of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[17] Michael Benedikt, Juliana Freire, and Patrice Godefroid. VeriWeb: Automatically testing dynamic Web sites. In *WWW*, 2002.

[18] Marina Biberstein, Joseph Gil, and Sara Porat. Sealing, encapsulation, and mutability. In *ECOOP 2001 — Object-Oriented Programming, 15th European Conference*, pages 28–52, Budapest, Hungary, June 18–22, 2001.

[19] Adrian Birka and Michael D. Ernst. A practical type system and language for reference immutability. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2004)*, pages 35–49, Vancouver, BC, Canada, October 26–28, 2004.

[20] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: Automated testing based on Java predicates. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 123–133, Rome, Italy, July 22–24, 2002.

[21] John Boyland. Why we should not add `readonly` to Java (yet). In *FTfJP'2005: 7th Workshop on Formal Techniques for Java-like Programs*, Glasgow, Scotland, July 26, 2005.

[22] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP 2001 — Object-Oriented Programming, 15th European Conference*, pages 2–27, Budapest, Hungary, June 18–22, 2001.

[23] David Brumley, Juan Caballero, Zhenkai Liang, James Newsome, and Dawn Song. Towards automatic discovery of deviations in binary implementations with applications to error detection and fingerprint generation. In *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007.

[24] Lilian Burdy, Yoonsik Cheon, David Cok, Michael D. Ernst, Joe Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212–232, June 2005.

[25] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, 2008.

[26] Cristian Cadar and Dawson R. Engler. Execution generated test cases: How to make systems code crash itself. In *SPIN*, 2005.

[27] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *CCS*, 2006.

[28] Néstor Cataño and Marieke Huisman. Chase: a static checker for JML's *assignable* clause. In *VMCAI'03, Fourth International Conference on Verification, Model Checking and Abstract Interpretation*, pages 26–40, New York, New York, January 9–11, 2003.

[29] Huo Yan Chen, T. H. Tse, and T. Y. Chen. TACCLE: A methodology for object-oriented software testing at the class and cluster levels. *IEEE Transactions on Software Engineering*, 10(1):56–109, 2001.

[30] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 232–245, Charleston, SC, January 1993.

[31] Jong-Deok Choi and Harini Srinivasan. Deterministic replay of Java multithreaded applications. In *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools (SPDT '98)*, pages 48–59, Welches, OR, USA, August 3–4, 1998.

[32] James Clause and Alessandro Orso. A technique for enabling and supporting debugging of field failures. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, pages 261–270, Minneapolis, MN, USA, May 23–25, 2007.

[33] Lars R. Clausen. A Java bytecode optimizer using side-effect analysis. *Concurrency: Practice and Experience*, 9(11):1031–1045, 1997.

[34] Holger Cleve and Andreas Zeller. Locating causes of program failures. In *ICSE'05, Proceedings of the 27th International Conference on Software Engineering*, pages 342–351, St. Louis, MO, USA, May 18–20, 2005.

[35] Conformiq. Conformiq test generator. `http://www.conformiq.com/`.

[36] Jonathan E. Cook and Alexander L. Wolf. Discovering models of software processes from event-based data. *ACM Transactions on Software Engineering and Methodology*, 7(3):215–249, July 1998.

[37] Keith D. Cooper and Ken Kennedy. Interprocedural side-effect analysis in linear time. In *PLDI 1988, Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, Atlanta, GA, USA, June 1988.

[38] Telmo Luis Correa Jr., Jaime Quinonez, and Michael D. Ernst. Tools for enforcing and inferring reference immutability in Java. In *Companion to Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, pages 866–867, Montréal, Canada, October 23–25, 2007.

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

[40] Christoph Csallner and Yannis Smaragdakis. Check 'n' Crash: Combining static checking and testing. In *ICSE'05, Proceedings of the 27th International Conference on Software Engineering*, pages 422–431, St. Louis, MO, USA, May 18–20, 2005.

[41] Christoph Csallner and Yannis Smaragdakis. DSD-Crasher: A hybrid analysis tool for bug finding. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 245–254, Portland, ME, USA, July 18–20, 2006.

[42] Christoph Csallner, Nikolai Tillmann, and Yannis Smaragdakis. DySy: Dynamic symbolic execution for invariant inference. In *ICSE*, 2008.

[43] Valentin Dallmeier, Christian Lindig, Andrzej Wasylkowski, and Andreas Zeller. Mining object behavior with ADABU. In *WODA 2006: Workshop on Dynamic Analysis*, pages 17–24, Shanghai, China, May 23, 2006.

[44] Daniela A. S. de Oliveira, Jedidiah R. Crandall, Gary Wassermann, S. Felix Wu, Zhendong Su, and Frederic T. Chong. ExecRecorder: VM-based full-system replay for attack analysis and system recovery. In *ASID '06: Proceedings of the 1st Workshop on Architectural and System Support for Improving Software Dependability*, pages 66–71, San Jose, CA, USA, October 21, 2006.

[45] D. Dean and D. Wagner. Intrusion detection via static analysis. In *Symposium on Research in Security and Privacy*, May 2001.

[46] Jeffrey Dean, David Grove, and Craig Chambers. Optimization of object-oriented programs using static class hierarchy analysis. In *ECOOP '95, the 9th European Conference on Object-Oriented Programming*, pages 77–101, Åarhus, Denmark, August 7–11, 1995.

[47] Claudio Demartini, Radu Iosif, and Riccardo Sisto. A deadlock detection tool for concurrent Java programs. *Software — Practice and Experience*, 29(7):577–603, June 1999.

[48] Brian Demsky and Martin Rinard. Role-based exploration of object-oriented programs. In *ICSE'02, Proceedings of the 24th International Conference on Software Engineering*, pages 313–324, Orlando, Florida, May 22–24, 2002.

[49] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, October 2005.

[50] Amer Diwan, J. Eliot B. Moss, and Kathryn S. McKinley. Simple and effective analysis of statically-typed object-oriented programs. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 292–305, San Jose, CA, USA, October 6–10, 1996.

[51] José Javier Dolado, Mark Harman, Mari Carmen Otero, and Lin Hu. An empirical investigation of the influence of a type of side effects on program comprehension. *IEEE Transactions on Software Engineering*, 29(7):665–670, July 2003.

[52] Roong-Ko Doong and Phyllis G. Frankl. Case studies on testing object-oriented programs. In *Proceedings of the ACM SIGSOFT '91 Fourth Symposium on Testing, Analysis, and Verification (TAV4)*, pages 165–177, Victoria, BC, Canada, October 8–10, 1991.

[53] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *USENIX 5th Symposium on OS Design and Implementation*, pages 211–224, Boston, MA, USA, December 9–11, 2002.

[54] Sebastian Elbaum, Kalyan-Ram Chilakamarri, Marc Fisher, and Gregg Rothermel. Web application characterization through directed requests. In *WODA*, 2006.

[55] Sebastian Elbaum, Hui Nee Chin, Matthew B. Dwyer, and Jonathan Dokulil. Carving differential unit test cases from system test cases. In *Proceedings of the ACM SIGSOFT 14th Symposium on the Foundations of Software Engineering (FSE 2006)*, pages 253–264, Portland, OR, USA, November 7–9, 2006.

[56] Sebastian Elbaum, Srikanth Karre, Gregg Rothermel, and Marc Fisher. Leveraging user-session data to support Web application testing. *IEEE Trans. Softw. Eng.*, 31(3), 2005.

[57] Michael Emmi, Rupak Majumdar, and Koushik Sen. Dynamic test input generation for database applications. In *ISSTA*, 2007.

[58] Michael D. Ernst. Type annotations specification (jsr 308). `http://pag.csail.mit.edu/jsr308/`, September 12, 2008.

[59] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001. A previous version appeared in *ICSE '99, Proceedings of the 21st International Conference on Software Engineering*, pages 213–224, Los Angeles, CA, USA, May 19–21, 1999.

[60] Marc Fisher, Sebastian G. Elbaum, and Gregg Rothermel. Dynamic characterization of Web application interfaces. In *FASE*, 2007.

[61] Jeffrey S. Foster, Manuel Fähndrich, and Alexander Aiken. A theory of type qualifiers. In *PLDI 1999, Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pages 192–203, Atlanta, GA, USA, May 1–4, 1999.

[62] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 2000.

[63] Simson Garfinkel. History's worst software bugs, 2005.

[64] Dennis Geels, Gautam Altekar, Scott Shenker, and Ion Stoica. Replay debugging for distributed applications. In *Proceedings of the 2006 USENIX Annual Technical Conference*, pages 289–300, Boston, MA, USA, June 1–3, 2006.

[65] Patrice Godefroid. Compositional dynamic test generation. In *Proceedings of the 34rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 47–54, Nice, France, January 17–19, 2007.

[66] Patrice Godefroid, Adam Kieżun, and Michael Y. Levin. Grammar-based whitebox fuzzing. In *PLDI*, 2008.

[67] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *PLDI 2005, Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation*, Chicago, IL, USA, June 13–15, 2005.

[68] Patrice Godefroid, Michael Y. Levin, and David Molnar. Automated whitebox fuzz testing. In *NDSS*, 2008.

[69] David Greenfieldboyce and Jeffrey S. Foster. Type qualifier inference for Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2007)*, pages 321–336, Montréal, Canada, October 23–25, 2007.

[70] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 112–122, Rome, Italy, July 22–24, 2002.

[71] Philip Jia Guo. A scalable mixed-level approach to dynamic analysis of C and C++ programs. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 5, 2006.

[72] William G. J. Halfond and Alessandro Orso. Improving test case generation for Web applications using automated interface discovery. In *ESEC-FSE*, 2007.

[73] A. Hartman and K. Nagin. The AGEDIS tools for model based testing. In *ISSTA 2004, Proceedings of the 2004 International Symposium on Software Testing and Analysis*, pages 129–132, Boston, MA, USA, July 12–14, 2004.

[74] Klaus Havelund and Thomas Pressburger. Model checking Java programs using Java PathFinder. *Software Tools for Technology Transfer*, 2(4):366–381, 2000.

[75] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'01)*, pages 54–61, Snowbird, Utah, USA, June 18–19, 2001.

[76] Daniel Hoffman and Paul Strooper. Classbench: a framework for automated class testing. *Software: Practice and Experience*, 27(5):573–597, 1997.

[77] John Hogg. Islands: Aliasing protection in object-oriented languages. In *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 271–285, Phoeniz, AZ, USA, October 1991.

[78] Gerard J. Holzmann. The model checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[79] Yao-Wen Huang, Fang Yu, Christian Hang, Chung-Hung Tsai, D. T. Lee, and Sy-Yen Ku. Verifying Web applications using bounded model checking. In *Proceedings of International Conference on Dependable Systems and Networks*, 2004.

[80] Atsushi Igarashi, Benjamin C. Pierce, and Philip Wadler. Featherweight Java: a minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, May 2001.

[81] Kobi Inkumsah and Tao Xie. Evacon: a framework for integrating evolutionary and concolic testing for object-oriented programs. In *ASE*, 2007.

[82] Martin Johns and Christian Beyerlein. SMask: preventing injection attacks in Web applications by approximating automatic data/code separation. In *SAC*, 2007.

[83] James A. Jones and Mary Jean Harrold. Empirical evaluation of the Tarantula automatic fault-localization technique. In *ASE 2005: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 273–282, Long Beach, CA, USA, November 9–11, 2005.

[84] Nenad Jovanovic, Christopher Kruegel, and Engin Kirda. Pixy: A static analysis tool for detecting Web application vulnerabilities (short paper). In *Security and Privacy*, 2006.

[85] Adam Kieżun, Philip Guo, Karthick Jayaraman, and Michael Ernst. Automatic creation of SQL injection and cross-site scripting attacks. In *Proceedings of International Conference of Software Engineering (ICSE)*, 2009.

[86] Günter Kniesel and Dirk Theisen. JAC — access right based encapsulation for Java. *Software: Practice and Experience*, 31(6):555–576, 2001.

[87] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *PLDI 1992, Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, San Francisco, Calif., June 1992.

[88] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *PLDI 1993, Proceedings of the SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, Albuquerque, NM, USA, June 23–25, 1993.

[89] Anatole Le, Ondřej Lhoták, and Laurie Hendren. Using inter-procedural side-effect information in JIT optimizations. In *Compiler Construction: 14th International Conference, CC 2005*, pages 287–304, Edinburgh, Scotland, April 2005.

[90] T. J. LeBlanc and J. M. Mellor-Crummey. Debugging parallel programs with instant replay. *IEEE Transactions on Computers*, 36(4):471–482, April 1987.

[91] Andreas Leitner, Ilinca Ciupa, Manuel Oriol, Bertrand Meyer, and Arno Fiva. Contract Driven Development = Test Driven Development – writing test cases. In *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 425–434, Dubrovnik, Croatia, September 5–7, 2007.

[92] Nancy Leveson and Clark S. Turner. An investigation of the therac-25 accidents. *IEEE Computer*, 26(7):18–41, 1993.

[93] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE*, 2007.

[94] Rupak Majumdar and Ru-Gang Xu. Directed test generation using symbolic grammars. In *ASE*, 2007.

[95] Leonardo Mariani and Mauro Pezzè. Behavior capture and test: Automated analysis of component integration. In *International Conference on Engineering of Complex Computer Systems*, pages 292–301, Shanghai, China, June 16-20, 2005.

[96] Sean McAllister, Engin Kirda, and Christopher Kruegel. Leveraging user interactions for in-depth testing of web applications. In *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 191–210, Berlin, Heidelberg, 2008. Springer-Verlag.

[97] Samir V. Meghani and Michael D. Ernst. Determining legal method call sequences in object interfaces, May 2003.

[98] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to and side-effect analyses for Java. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 1–11, Rome, Italy, July 22–24, 2002.

[99] Yasuhiko Minamide. Static approximation of dynamically generated Web pages. In *WWW*, 2005.

[100] Ghassan Misherghi and Zhendong Su. HDD: hierarchical delta debugging. In *ICSE*, 2006.

[101] Gail C. Murphy, David Notkin, and Erica S.-C. Lan. An empirical study of static call graph extractors. In *Proceedings of the 18th International Conference on Software Engineering*, pages 90–99, March 1996.

[102] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Continuously recording program execution for deterministic replay debugging. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture*, pages 284–295, Madison, WI, USA, June 6–8, 2005.

[103] Satish Narayanasamy, Gilles Pokam, and Brad Calder. BugNet: Recording application-level execution for deterministic replay debugging. *IEEE Micro*, 26(1):100–109, 2006.

[104] James Noble, Jan Vitek, and John Potter. Flexible alias protection. In *ECOOP '98, the 12th European Conference on Object-Oriented Programming*, pages 158–185, Brussels, Belgium, July 20-24, 1998.

[105] ObjectWeb Consortium. ASM - Home Page, 2007. `http://asm.objectweb.org/`.

[106] Robert O'Callahan. Personal communication, 2008.

[107] Robert O'Callahan and Jong-Deok Choi. Hybrid dynamic data race detection. In *Proceedings of the 9th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 167–178, San Diego, CA, USA, July 11–13 2003.

[108] Catherine Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *QoSA/SOQUA*, pages 242–256, September 2005.

[109] Alessandro Orso and Bryan Kennedy. Selective capture and replay of program executions. In *WODA 2005: Workshop on Dynamic Analysis*, pages 1–7, St. Louis, MO, USA, May 17, 2005.

[110] Carlos Pacheco and Michael D. Ernst. Eclat: Automatic generation and classification of test inputs. In *ECOOP 2005 — Object-Oriented Programming, 19th European Conference*, pages 504–527, Glasgow, Scotland, July 27–29, 2005.

[111] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *ICSE'07, Proceedings of the 29th International Conference on Software Engineering*, pages 75–84, Minneapolis, MN, USA, May 23–25, 2007.

[112] Matthew M. Papi, Mahmood Ali, Telmo Luis Correa Jr., Jeff H. Perkins, and Michael D. Ernst. Practical pluggable types for Java. In *ISSTA 2008, Proceedings of the 2008 International Symposium on Software Testing and Analysis*, pages 201–212, Seattle, WA, USA, July 22–24, 2008.

[113] Igor Pechtchanski and Vivek Sarkar. Immutability specification and its applications. In *Joint ACM-ISCOPE Java Grande Conference*, pages 202–211, Seattle, WA, November 3–5, 2002.

[114] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[115] Tadeusz Pietraszek and Chris Vanden Berghe. Defending against injection attacks through context-sensitive string evaluation. In *RAID*, 2005.

[116] Marina Polishchuk, Ben Liblit, and Chloë Schulze. Dynamic heap type inference for program understanding and debugging. In *Proceedings of the 34rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 39–46, Nice, France, January 17–19, 2007.

[117] Sara Porat, Marina Biberstein, Larry Koved, and Bilba Mendelson. Automatic detection of immutable fields in Java. In *CASCON*, Mississauga, Ontario, Canada, November 13–16, 2000.

[118] Jaime Quinonez. Inference of reference immutability in Java. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, May 2008.

[119] Jaime Quinonez, Matthew S. Tschantz, and Michael D. Ernst. Inference of reference immutability. In *ECOOP 2008 — Object-Oriented Programming, 22nd European Conference*, pages 616–641, Paphos, Cyprus, July 9–11, 2008.

[120] Chrislain Razafimahefa. A study of side-effect analyses for Java. Master's thesis, School of Computer Science, McGill University, Montreal, Canada, December 1999.

[121] Reactive Systems, Inc. Reactis. `http://www.reactive-systems.com/`.

[122] NIST report. Software errors cost u.s. economy $59.5 billion annually, 2002.

[123] Filippo Ricca and Paolo Tonella. Analysis and testing of Web applications. In *ICSE*, 2001.

[124] Atanas Rountev. Precise identification of side-effect-free methods in Java. In *ICSM 2004, Proceedings of the International Conference on Software Maintenance*, pages 82–91, Chicago, Illinois, September 12–14, 2004.

[125] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java based on annotated constraints. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2001)*, pages 43–55, Tampa Bay, FL, USA, October 14–18, 2001.

[126] Atanas Rountev and Barbara G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. In *Compiler Construction: 10th International Conference, CC 2001*, pages 20–36, Genova, Italy, April 2001.

[127] Erik Ruf. Context-insensitive alias analysis reconsidered. In *PLDI 1995, Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, La Jolla, CA, USA, June 1995.

[128] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Transactions on Programming Languages and Systems*, 23(2):105–186, March 2001.

[129] David Saff, Shay Artzi, Jeff H. Perkins, and Michael D. Ernst. Automatic test factoring for Java. In *ASE 2005: Proceedings of the 20th Annual International Conference on Automated Software Engineering*, pages 114–123, Long Beach, CA, USA, November 9–11, 2005.

[130] Alexandru Sălcianu. *Pointer analysis for Java programs: Novel techniques and applications*. PhD thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, September 2006.

[131] Alexandru Sălcianu and Martin C. Rinard. Purity and side-effect analysis for Java programs. In *VMCAI'05, Sixth International Conference on Verification, Model Checking and Abstract Interpretation*, pages 199–215, Paris, France, January 17–19, 2005.

[132] Stefan Savage, Michael Burrows, Greg Nelson, Patrick Sobalvarro, and Thomas Anderson. Eraser: A dynamic data race detector for multi-threaded programs. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles*, pages 27–37, St. Malo, France, December 5–8, 1997.

[133] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A concolic unit testing engine for C. In *ESEC/FSE 2005: Proceedings of the 10th European Software Engineering Conference and the 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 263–272, Lisbon, Portugal, September 7–9, 2005.

[134] Mats Skoglund and Tobias Wrigstad. A mode system for read-only references in Java. In *FTfJP'2001: 3rd Workshop on Formal Techniques for Java-like Programs*, Glasgow, Scotland, June 18, 2001.

[135] Sara Sprenkle, Emily Gibson, Sreedevi Sampath, and Lori Pollock. Automated replay and failure detection for Web applications. In *ASE*, 2005.

[136] Sudarshan M. Srinivasan, Srikanth Kandula, Christopher R. Andrews, and Yuanyuan Zhou. Flashback: A lightweight extension for rollback and deterministic replay for software debugging. In *Proceedings of the 2004 USENIX Annual Technical Conference*, pages 29–44, Boston, MA, USA, June 28–July 2, 2004.

[137] John Steven, Pravir Chandra, Bob Fleck, and Andy Podgurski. jRapture: A capture/replay tool for observation-based testing. In *ISSTA 2000, Proceedings of the 2000 International Symposium on Software Testing and Analysis*, pages 158–167, Portland, OR, USA, August 22–25, 2000.

[138] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley, Boston, MA, special edition, 2000.

[139] Zhendong Su and Gary Wassermann. The essence of command injection attacks in Web applications. In *POPL*, 2006.

[140] Frank Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3):121–189, 1995.

[141] Frank Tip and Jens Palsberg. Scalable propagation-based call graph construction algorithms. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2000)*, pages 281–293, Minneapolis, MN, USA, October 15–19, 2000.

[142] Oksana Tkachuk and Matthew B. Dwyer. Adapting side effects analysis for modular program model checking. In *ESEC/FSE 2003: Proceedings of the 9th European Software Engineering Conference and the 11th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 188–197, Helsinki, Finland, September 3–5, 2003.

[143] Aaron Tomb, Guillaume Brat, and Willem Visser. Variably interprocedural program analysis for runtime error detection. In *ISSTA 2007, Proceedings of the 2007 International Symposium on Software Testing and Analysis*, pages 97–107, London, UK, July 10–12, 2007.

[144] Jan Tretmans and Ed Brinksma. TorX: Automated model based testing. In *1st European Conference on Model Driven Software Engineering*, pages 31–43, 2003.

[145] Matthew S. Tschantz. Javari: Adding reference immutability to Java. Master's thesis, MIT Department of Electrical Engineering and Computer Science, Cambridge, MA, August 2006.

[146] Matthew S. Tschantz and Michael D. Ernst. Javari: Adding reference immutability to Java. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA 2005)*, pages 211–230, San Diego, CA, USA, October 18–20, 2005.

[147] Christopher D. Turner and David J. Robson. The state-based testing of object-oriented programs. In *Proceedings of the Conference on Software Maintenance*, pages 302–310, Montréal, Quebec, Canada, September 1993.

[148] Mandana Vaziri, Frank Tip, Stephen Fink, and Julian Dolby. Declarative object identity using relation types. In *ECOOP 2007 — Object-Oriented Programming, 21st European Conference*, pages 54–78, Berlin, Germany, August 1-3, 2007.

[149] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for Java containers using state matching. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 37–48, Portland, ME, USA, July 18–20, 2006.

[150] Gary Wassermann and Zhendong Su. Sound and precise analysis of Web applications for injection vulnerabilities. In *PLDI*, 2007.

[151] Gary Wassermann and Zhendong Su. Static detection of cross-site scripting vulnerabilities. In *ICSE*, 2008.

[152] Gary Wassermann, Dachuan Yu, Ajay Chander, Dinakar Dhurjati, Hiroshi Inamura, and Zhendong Su. Dynamic test input generation for web applications. In *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 249–260, 2008.

[153] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[154] John Whaley, Michael Martin, and Monica Lam. Automatic extraction of object-oriented component interfaces. In *ISSTA 2002, Proceedings of the 2002 International Symposium on Software Testing and Analysis*, pages 218–228, Rome, Italy, July 22–24, 2002.

[155] Tao Xie. Augmenting automatically generated unit-test suites with regression oracle checking. In *ECOOP 2006 — Object-Oriented Programming, 20th European Conference*, pages 380–403, Nantes, France, July 5–7, 2006.

[156] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE 2004: Proceedings of the 19th Annual International Conference on Automated Software Engineering*, pages 196–205, Linz, Australia, September 22–24, 2004.

[157] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 365–381, Edinburgh, UK, April 4–8, 2005.

[158] Yichen Xie and Alex Aiken. Static detection of security vulnerabilities in scripting languages. In *USENIX-SS*, 2006.

[159] Haiying Xu, Christopher J. F. Pickett, and Clark Verbrugge. Dynamic purity analysis for Java programs. In *ACM SIGPLAN/SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE 2007)*, pages 75–82, San Diego, CA, USA, June 13–14, 2007.

[160] Min Xu, Rastislav Bodik, and Mark D. Hill. A “flight data recorder” for enabling full-system multiprocessor deterministic replay. In *Proceedings of the 30th Annual International Symposium on Computer Architecture*, pages 122–135, San Diego, CA, USA, June 9–11, 2003.

[161] Jinlin Yang and David Evans. Automatically inferring temporal properties for program evolution. In *Fifteenth International Symposium on Software Reliability Engineering*, pages 340–351, Saint-Malo, France, November 3–5, 2004.

[162] Andreas Zeller. Yesterday, my program worked. Today, it does not. Why? In *Proceedings of the 7th European Software Engineering Conference and the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 253–267, Toulouse, France, September 6–9, 1999.

[163] Andreas Zeller. Isolating cause-effect chains from computer programs. In *Proceedings of the ACM SIGSOFT 10th International Symposium on the Foundations of Software Engineering (FSE 2002)*, pages 1–10, Charleston, SC, November 20–22, 2002.

[164] Yoav Zibin, Alex Potanin, Mahmood Ali, Shay Artzi, Adam Kieżun, and Michael D. Ernst. Object and reference immutability using Java generics. In *ESEC/FSE 2007: Proceedings of the 11th European Software Engineering Conference and the 15th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 75–84, Dubrovnik, Croatia, September 5–7, 2007.

[165] Federico Zoufaly. Web standards and search engine optimization (seo) – does google care about the quality of your markup?, 2008.