Directed Random Testing

by

Carlos Pacheco

B.S., University of Texas at Austin (2001)
S.M., Massachusetts Institute of Technology (2005)

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

Doctor of Philosophy

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 2009

Author . . . .
Department of Electrical Engineering and Computer Science
February 25, 2009

Certified by . . . .
Daniel N. Jackson
Professor of Electrical Engineering and Computer Science
Thesis Supervisor

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

Directed Random Testing

by

Carlos Pacheco

Submitted to the Department of Electrical Engineering and Computer Science
on February 25, 2009, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

Random testing can quickly generate many tests, is easy to implement, scales to large software applications, and reveals software errors. But it tends to generate many tests that are illegal or that exercise the same parts of the code as other tests, thus limiting its effectiveness. Directed random testing is a new approach to test generation that overcomes these limitations, by combining a bottom-up generation of tests with runtime guidance. A directed random test generator takes a collection of operations under test and generates new tests incrementally, by randomly selecting operations to apply and finding arguments from among previously-constructed tests. As soon as it generates a new test, the generator executes it, and the result determines whether the test is redundant, illegal, error-revealing, or useful for generating more tests. The technique outputs failing tests pointing to potential errors that should be corrected, and passing tests that can be used for regression testing. The thesis also contributes auxiliary techniques that post-process the generated tests, including a simplification technique that transforms a failing test into a smaller one that better isolates the cause of failure, and a branch-directed test generation technique that aims to increase the code coverage achieved by the set of generated tests.

Applied to 14 widely-used libraries (including the Java JDK and the core .NET framework libraries), directed random testing quickly reveals many serious, previously unknown errors in the libraries. And compared with other test generation tools (model checking, symbolic execution, and traditional random testing), it reveals more errors and achieves higher code coverage. In an industrial case study, a test team at Microsoft using the technique discovered in fifteen hours of human effort as many errors as they typically discover in a person-year of effort using other testing methods.

Thesis Supervisor: Daniel N. Jackson
Title: Professor of Electrical Engineering and Computer Science

Acknowledgments

I am grateful to the many people who have contributed to this thesis with their support, collaboration, mentorship, and friendship. Special thanks go to Daniel Jackson and Michael Ernst, my two advisors. Daniel provided invaluable insights and guidance in all aspects of my work. He pushed me to make my ideas more precise, to search for simplicity, and to bring out the excitement in my work. Daniel's positive attitude, sensitivity, and unwavering belief in me made working with him a true pleasure. Every time I left one of our meetings, I felt energized and ready to attack any problem, however unsurmountable it appeared beforehand. I am equally thankful to Michael Ernst, who collaborated with me on much of my research and was a tireless advocate of my work. Mike's teaching was key during my years at MIT—he taught me a great deal about research, writing, empirical work, and spreading the word about my ideas. Working with Mike made me a better researcher and writer.

My thesis committee members, Srini Devadas and Armando Solar-Lezama, provided valuable feedback on the thesis, and selfless support in the process. Srini's excellent questions improved the final version of the document, his advice helped me keep the ship steady, and his role as a trusted mentor during my last years at MIT made an enormous difference at crucial times. Armando's door was always open, and a number of aspects of my dissertation and oral defense benefited from his input.

I have been privileged to have great mentors outside MIT as well. Michael Merritt from the AT&T Labs fellowship program was always a phone call away, and helped me navigate my graduate school experience. J Moore, my undergraduate advisor, has been an important role model since my days at UT Austin. I am grateful to J for his encouragement while at UT and MIT.

During my Ph.D. studies, I participated in two wonderful and formative internships at Microsoft Research. Tom Ball and Shuvendu Lahiri were not only colleagues, but also champions of my work. Working with Scott Wadsworth using my techniques to find errors in .NET was a great learning experience. My many discussions with Scott, who has deep knowledge and experience testing complex software, motivated me to develop techniques that have impact in the real world. Christoph Csallner, Nachi Nagappan, Madan Musuvathi, Nikolai Tillmann, and the members of the Software Reliability Research and Foundations of Software Engineering group at Microsoft Research made my Microsoft internships intellectually and personally gratifying.

My colleagues and friends at MIT (and beyond) made my years as a graduate student rewarding. Shay Artzi, Marcelo d'Amorim, Adam Kieżun, Darko Marinov, and Jeff Perkins were fantastic collaborators; working on exciting projects with them was one of the highlights of my time at MIT. I am thankful to Jeff for providing a friendly ear and teaching me about effective project management, a skill I am sure will be useful in my career. Derek Rayside, Vijay Ganesh, Robert Seater, and Emina Torlak gave me timely comments as I iterated on my dissertation and oral defense. I shared many interesting conversations with Jonathan Edwards, Felix Chang, Greg

Dennis, Eunsuk Kang, Stephen McCamant, David Saff, Mana Taghdiri, and Amy Williams—they all made made my days in the Stata Center enjoyable. Joe Near, Aleksandar Milicevic, and Rishabh Singh kindly evaluated Randoop and filled out questionnaires about their experiences, from which I learned more about the tool's strengths and weaknesses. Maria Rebelo, Mary McDavitt, Peggy Carney, Janet Fischer and Marilyn Pierce helped me with financial and organizational matters, big and small. The friendship of my dear friends Ajit Dash, Tom Giove, and Stan Zanarotti made Boston home, the high points more joyful, and helped me gain perspective during the natural low points.

Much of my stay at MIT was funded by a Paul and Daisy Soros fellowship for new Americans, a National Science Foundation graduate research fellowship, an Akamai presidential fellowship, and an AT&T Labs fellowship. I am truly grateful to all the folks in these organizations. Their financial support allowed me to concentrate on research.

My last but deepest thanks go to my family, without whose constant stream of love and support I would not be here. My always radiant, super-woman mom and my kind, gifted sister are the two most amazing women I know; they show me by example what is important in life. My wonderful dad's words of wisdom have always helped me grow as a person. My partner Itay has walked with me through many challenges with unconditional support and patience. They are the best family I could have asked for.

Contents

Chapter 1

Introduction

Software companies spend vast resources testing the software they build, to find and remove as many errors as possible before they ship their products. An estimated thirty to ninety percent of the total cost of developing software goes into testing [12, 33]. At Microsoft, for example, there is roughly one test engineer for every software developer. Still, software remains full of errors, resulting in billions of dollars in losses across the economy: A 2002 report by the National Institute of Standards and Technology estimates the yearly cost of inadequate software testing on the US economy to be approximately sixty billion dollars [33]. One third of the cost is borne by software companies, who must correct defects and distribute fixes, and two thirds by end users, who suffer decreased productivity due to software malfunctions; for example, software errors in the finance industry cause monetary losses due to aborted and delayed transactions, system shut downs, and manual effort in correcting processing errors.

Part of the reason for the cost and inadequacy of testing is that it remains a largely manual process, even when many aspects of the process could benefit from automation. This thesis presents a technique that automates the creation of test cases, a primary testing activity. *Directed random testing* mechanically creates a set of test cases that the engineer can use to start a test suite or complement an existing one. The technique combines random generation of test inputs with systematic pruning of the input space to avoid generating illegal and equivalent inputs.

Directed random testing can complement manual testing by automatically generating test cases that cover scenarios an engineer may have missed. Our experiments and case studies show that the technique can reveal errors that evade manual testing and other automated test case generation techniques, and that directed random testing can increase the productivity of a team of test engineers working in an industrial environment by orders of magnitude: using directed random testing, a test team at Microsoft discovered, in fifteen hours of human effort, as many errors as are typically discovered in a man-year of effort, using other testing methods.

1.1 Problem Description and Previous Approaches

1.1.1 Test Suites are Valuable

Testing a software application involves executing the application on a set of inputs, and evaluating whether it behaves as expected on the inputs. To do this, a test engineer creates a set of test cases, or *test suite.* Each test case consists of an *input* that drives execution of the software in a specific scenario, and an *oracle* that checks for expected program behavior.

Software development organizations use test suites to assess the reliability of the software they develop; a software artifact that leads to a large number of failing test cases is bound to be unreliable in practice. Test suites are also valuable for discovering and fixing errors, because failing test cases offer clues about their location in the code. When testing a software artifact, a test engineer aims to create a test suite that is as comprehensive as possible, i.e. a test suite that can reveal as many errors as possible in the software under test.

1.1.2 Creating a Comprehensive Test Suite is Challenging

The input space of many realistic software artifacts is so large as to be practically infinite, and creating a set of test cases representative of the important scenarios is an intellectually challenging and time consuming task. Test engineers use a number of methods [3] to help them create an adequate set of test cases (e.g. partition testing), as well as coverage metrics that give some measure of the adequacy of a test suite (e.g. statement or branch coverage) after it has been created. And yet, most test suites miss important cases.

Consider for example the widely-used Java Development Kit [40], a set of libraries for Java spanning several hundred thousand lines of code. Over the years, Sun has developed a test suite for the JDK called the Java Compatibility Kit which, according to Sun, is an "extensive test suite to ensure compatible implementations of the Java Platform" [56]. Despite significant effort on Sun's part to create a comprehensive test suite, the test suite lacks many test cases that miss errors present in the JDK, as evidenced by the thousands of JDK "bug reports" in Sun's bug database [88] and by our experience using directed random testing to reveal errors in the libraries (Chapters 8 and 9). Our aim is not to pick on Sun; all software development companies face the problem of inadequate test suites. For example, Chapter 9 describes a case study in which a test team at Microsoft discovered missing important test cases in a test suite for a core component of the .NET framework [74].

Automated test generation techniques can create many more inputs than an engineer would have time to create by hand, revealing important scenarios missed by manual efforts. Two well-established approaches to automated test generation are random and exhaustive testing. *Random testing* generates inputs at random from an input grammar. Despite its naïve generation strategy and lack of completeness, random testing is surprisingly effective in producing error-revealing test cases [46]. At the other end of the spectrum, *exhaustive testing* generates all inputs up to some input size bound [21, 4, 87]. To improve efficiency, many exhaustive generators use "pruning" techniques that prevent the generation of *equivalent inputs* on which the software under test behaves identically, and *illegal inputs* that the software is not required to handle. When feasible, exhaustive testing is attractive because it provides complete input coverage up to the bound. The downside of the approach is its limited applicability due to the large input space of many software artifacts.

1.2 Directed Random Testing

1.2.1 Key Idea: Generate Randomly, Prune Systematically

For many programs, exhaustive testing is infeasible, even for small input size bounds. Nevertheless, the idea of pruning can be applied to increase the effectiveness of non-exhaustive approaches such as random testing. *Directed random testing* combines random generation with input pruning.

Directed random testing generates tests for software whose input data can be built compositionally from a set of operations. Examples include container data structures like linked lists or hash tables that can be built by adding and removing items; XML trees built from smaller XML subtrees (e.g. a table consisting of table rows); or mathematical objects like matrices or polynomials that can be built from similar objects by addition and multiplication.

We call the operations used to create input data *builder operations*. Directed random testing generates new inputs incrementally, by applying builder operations to previously-generated inputs. At each step, the technique creates a new input by randomly selecting a builder operation, finding arguments from among previous inputs. The technique executes new inputs and uses the result of the execution to heuristically determine if an input is illegal or if two inputs are equivalent. It prunes the input space by avoiding using such inputs as arguments to further operations.

1.2.2 Example

Consider the task of testing a procedure that adds a polynomial and a monomial; for example, given the polynomial $2x^3+1$ and the monomial x^2, the procedure returns the polynomial $2x^3+x^2+1$. The procedure takes as inputs an instance of a data type `Poly` that represents a polynomial, and an instance of a data type `Mono` that represents a monomial. Instances of this data type can be created using the operations and related data types shown (as Java constructs) in Figure 1-1. The data types shown are based on a real programming assignment given to undergraduate computer science students at MIT.

Creating polynomial and monomial values using these operations is a recursive task. For example, in order to create a polynomial, we must first create some monomials (`Mono` values), and then call the operation `Poly(Mono... elements)` on the monomial values, respecting the preconditions stipulated by the operation (e.g. that the monomials are sorted in decreasing-exponent order). Similarly, to create a mono-

```
public class Rat {

  // Creates a Rat representing n/d. Requires: d is not 0.
  public Rat(int n, int d) { ... }
}
public class Mono {

  // Creates a monomial with the specified coefficient
  // and exponent. Requires: c is not null, e is not negative.
  public Mono(Rat c, int e) { ... }
}
public class Poly {

  // Creates a new polynomial with the specified elements.
  // Requires: elements array is not null, elements are sorted
  // in order of decreasing exponent, and have non-zero coefficient.
  public Poly(Mono... elements) { ... }
}
```

Figure 1-1: Three classes implementing polynomials. For brevity, we do not show all methods declared in the classes and omit method bodies.

mial we must first create two values, a rational number (representing the coefficient) and an integer (representing the exponent), and then call the operation `Mono(Rat coeff, int exp)` on those values. We can represent this recursive process using *terms* over the operations; for example, the term

```
Poly(Mono(Rat(2,1),3),Mono(Rat(1,1),0))
```

corresponds to the polynomial $2x^3 + 1$. We can create inputs to the procedure by creating terms over builder operations, and evaluating the terms to obtain actual runtime values.

Random Testing

A random input generator generates random terms over the polynomial operations. There are two fundamental problems with this approach. First, without knowledge of each operation's preconditions, a random generator can generate many *illegal* terms; for example, a term that calls the `Poly` operation with an unsorted list of monomials. Illegal terms can result in arbitrary (but not error-revealing) behavior in the code under test: the procedure under test expects well-formed polynomials as input, and may behave unexpectedly given an invalid polynomial. Second, a random generator can produce many *equivalent* terms, i.e. terms evaluating to values that cause the procedure under test to follow the same execution path, adding nothing to the testing effort. For example, the `Rat` operation always reduces the rational number it returns, making terms like `Poly(Mono(Rat(2,1),3))` and `Poly(Mono(Rat(4,2),3))` (both representing $2x^3$) equivalent, since their values elicit identical behavior on the procedure under test.

Directed Random Testing

Directed random testing improves random testing by discarding illegal and equivalent terms whenever they are generated. The technique maintains a set of previously-generated terms called a *component set*. The component set initially contains only a set of "seed" terms, e.g. terms representing primitive values like `0`, `1`, etc. The generation proceeds in steps. At each step, the generator randomly selects a new term among the set of all terms achievable by applying a new operation to terms in the component set, but restricts the selection to terms that are legal and non-equivalent to existing terms. For example, the restricted set of legal and non-equivalent terms, given the component set {`1`, `-1`, `Rat(1,1)`} is { `Rat(1,-1)`, `Rat(-1,1)`, `Mono(Rat(1,1),1)`}. The set excludes `Rat(1,1)` because it is already in the component set, `Rat(-1,-1)` because it is equivalent to a term in the component set, and `Mono(Rat(1,1),-1)` because it is illegal.

By restricting the generation process, directed random testing prunes the input space. For example, the generator never creates the illegal polynomial term `Poly(Mono(Rat(1,1),-1))`, because the illegal sub-term `Mono(Rat(1,1),-1)` is never added to the component set. The generator never creates both equivalent terms `Mono(Rat(1,1),1)` and `Mono(Rat(-1,-1),1)` because only one of the two sub-terms `Rat(1,1)` and `Rat(-1,-1)` is ever added to the component set. Because terms are compositional (and assuming legality and equivalence are compositional as well), the generator may prune an infinite number of illegal (equivalent) terms from the search space by preventing a single illegal (equivalent) term from being added to the component set.

To detect illegal and equivalent terms, directed random testing executes the terms as it generates them, and uses the result of the execution to decided if they should be discarded and not added to the component set. For instance, a term whose execution leads to an exception is judged illegal, and two terms that produce the same result when used as arguments to observer procedures (e.g. an `equals` method) are flagged as equivalent. The strategy is fully automatic, but potentially unsound. Our evaluation shows that it is effective in practice.

1.3 Randoop

Randoop implements directed random testing for Java and .NET. ("Randoop" stands for "**Rand**omized Test Generator for **O**bject-**O**riented **P**rograms.") The tool takes as input a set of classes, extracts their method signatures, and uses the methods as builder operations to create test inputs. Randoop generates test inputs until a user-specified time limit expires, and outputs the set generated inputs as compilable code.

Randoop can be used as an end-to-end test case generator (inputs plus oracles) to test a collection of methods. In this usage model, Randoop considers the methods in the classes as being both builder operations *and* methods under test. As it executes operations, Randoop checks a set of object validity properties (built into the tool, and user-extensible), outputs property-violating inputs as failing tests, and property-

Error-revealing test case

```
// Fails on Sun 1.5.
public static void test1() {
   LinkedList l1 = new LinkedList();
   Object o1 = new Object();
   l1.addFirst(o1);
   TreeSet t1 = new TreeSet(l1);
   Set s1 =
    Collections.unmodifiableSet(t1);
   Assert.assertTrue(s1.equals(s1));
}
```

Regression test case

```
// Passes on Sun 1.5, fails on Sun 1.6
public static void test2() {
   BitSet b = new BitSet();
   Assert.assertEquals(64, b.size());
   b.clone();
   Assert.assertEquals(64, b.size());
}
```

Figure 1-2: Two test cases output by Randoop, which implements directed random testing.

satisfying inputs as regression tests. Figure 1-2 shows two examples of test cases generated by Randoop. The first test case shows a violation of reflexivity of `equals`, a property that all Java objects must satisfy. The test case shows a scenario that creates an invalid set `s1` that returns `false` for `s1.equals(s1)`. The second test case is a regression test case that violates no properties on version 1.5 of the Java JDK, but when executed against version 1.6 of the JDK, reveals a regression error that causes the same `BitSet` object to report a different size.

1.4 Evaluation

We have evaluated directed random testing with experiments and case studies that compare it against other techniques. Our results indicate that directed random testing can outperform random and exhaustive testing, in terms of coverage and error detection. For example, when compared with model checking and random testing on four complex data structures used previously in the literature [92], directed random testing achieves higher or equal coverage than the other techniques, and on average achieves this coverage in one third of the time as systematic testing techniques. On a set of 14 software component libraries (totaling more than 4,500 classes and 780KLOC), directed random testing reveals many previously-unknown errors that the other techniques miss.

We have assessed the cost effectiveness of Randoop by conducting an industrial case study. As part of the study, test engineers at Microsoft applied directed random testing to a critical component of the .NET architecture. Due to its complexity and high reliability requirements, the component had already been tested by 40 test engineers over five years, using manual and automated testing techniques. Nevertheless, directed random testing revealed errors in the component that eluded previous testing, and did so two orders of magnitude faster than a test engineer would on average. Directed random testing also revealed errors in other testing and analysis tools, and deficiencies in the development team's manual testing methodologies.

1.5 Contributions

The thesis makes the following contributions.

1. Directed random testing, a new randomized test generation technique whose goal is to preserve random testing's strengths (e.g. error-revealing effectiveness, scalability, and automation) while mitigating its deficiencies (e.g. its tendency to generate illegal and equivalent inputs). Directed random testing combines a bottom-up, incremental approach to generating test inputs with automated heuristic pruning strategies to avoid illegal and redundant inputs.

2. Auxiliary techniques to increase the quality of the inputs generated by directed random testing. *Replacement-based simplification* is a test simplification techniques that aims to help the engineer isolate the cause of a failure. It removes and replaces operations from a long failing test case to yield a smaller and simpler failing test case. *Branch-directed generation* aims to increase the code coverage achieved by directed random testing by monitoring the execution of inputs that fail to follow specific branches in the software under test, and modifying with the goal of covering those branches.

3. A tool architecture that implements directed random testing for object-oriented software. The RANDOOP test generator implements the architecture for object-oriented software implemented in Java and .NET.

4. An experimental evaluation of directed random testing measuring its effectiveness in revealing errors and achieving code coverage. The experiments compare directed random testing's effectiveness on these measures with random, exhaustive, and manual testing. The subject programs include previous data structure benchmarks and large Java and .NET applications.

5. A case study assessing directed random testing's cost effectiveness in an industrial setting, which often has very different environmental conditions than a research setting. The study describes the use of directed random testing by experienced test engineers at Microsoft to generate new tests cases for a core .NET component library.

Other researchers have used directed random testing as a component in new techniques for test generation, test selection, and dynamic analysis [9, 78, 25, 57, 58, 64]; as a basis of comparison with manual and automated approaches to test generation [10, 5, 53]; as an experimental subject program [27]; and as a source of test cases for evaluating techniques not related to test generation [85]. Figure 1-3 gives a brief summary of the uses and evaluations of directed random testing that we are aware of as of this writing. RANDOOP has also been used in software engineering courses at Caltech, the University of Illinois, and Universidade Federal de Pernambuco in Brazil, and has found industrial use at Microsoft and the Fraunhofer Center for Experimental Software Engineering, where it is being applied to discover errors in NASA software.

1.6 Thesis Organization

The thesis is logically divided into three parts.

- **Preliminaries.** Chapter 2 discusses basic concepts in software testing. The chapter describes the qualities of a useful test suite, the random and exhaustive approaches to automated test case generation, and unit testing.

- **Approach.** Chapters 3 through 7 describe directed random testing. Chapter 3 describes a formal framework for test generation and introduces top-down and bottom-up generation, two basic strategies for creating terms over builder operations. Chapter 4 describes directed random testing, which extends bottom-up generation by pruning illegal or equivalent inputs. Chapter 5 describes problems with terms and extends the technique to address these problems. Chapter 6 describes post-processing techniques that modify the inputs generated by directed random testing to make them simpler to debug and to increase the branch coverage they achieve. Chapter 7 describes the architecture of RANDOOP, the implementation of directed random testing for Java and .NET.

- **Evaluation and discussion.** Chapters 8 through 12 assess the effectiveness of directed random testing in comparison with manual testing and with other automated test generation techniques, using experiments (Chapter 8), case studies (Chapter 9), a discussion of directed random testing's strengths and limitations (Chapter 10), and a survey of related work (Chapter 11). Chapter 12 offers concluding remarks.

Reference	Description
Artzi et al. [9]	Used Randoop as an input generation component in a technique implementing an automated static/dynamic mutability analysis for Java.
Orso et al. [78]	Used Randoop as a component of a regression testing technique that dynamically analyzes the execution of test inputs on two code versions for behavioral differences.
d'Amorim et al. [25]	Extended Randoop to implement a new test generation technique that identifies code changes in a set of classes, and uses the information to limit the generation scope of random testing.
Jaygari et al. [57]	Extended Randoop with techniques that monitor code coverage for each method under test and alters the selection distribution of methods based on coverage information.
Jaygari et al. [58]	Used Randoop as an input generation component in a test generation technique that combines directed random testing and adaptive random testing [18].
Yu et al. [64]	Combined Randoop and the Racetrack [98] race condition/deadlock detector to generate test inputs and analyze their execution for errors. Revealed three serious errors in a released Microsoft product.
Bacchelli et al. [10]	Used Randoop in a case study evaluating the effectiveness of manual and automatic unit test generation techniques. Randoop revealed 7 errors in 120 hours of manual effort (17 hours/error), compared with 14 errors in 490 hours for manual testing (35 hours/error).
Andrews et al. [5]	As part of the evaluation of a test generation technique based on genetic algorithms, compared branch coverage achieved by Randoop, JPF [91], and the authors' tool. All three techniques achieve the same coverage. (But Randoop achieves it in equal or less time.)
Inkumsah et al. [53]	As part of evaluating a test generation technique based on symbolic execution and evolutionary testing, compared branch coverage achieved by Randoop and five other test generation tools. Of 13 classes tested, Randoop achieved the highest coverage for 6, better-than-average coverage in 6, and worse-than average coverage in 5.
Daniel et al. [27]	Used Randoop as a subject program, to evaluate a machine learning algorithm that predicts code coverage achieved by a test generation tool, using as features metrics of the software under test.
Rayside et al. [85]	Used Randoop to generate tests used to evaluate a code generation technique for `equals/hashCode`. Used the test cases to check that generated code preserves original behavior and fixes errors.

Figure 1-3: Recent uses and evaluations of directed random testing by other researchers. The first six entries describe techniques that use the technique as a component in other techniques. The next three entries use directed random testing as a basis for comparison in the evaluation of other test generation techniques. The last two entries use directed random testing to generate test cases used to evaluate other techniques.

Chapter 2

Software Testing

Software testing [3] is large and diverse field, reflecting the different requirements that software artifacts must satisfy (robustness, efficiency, functional correctness, backward compatibility, usability, etc.), the different activities involved in testing (writing down requirements, designing test plans, creating test cases, executing test cases, checking results, isolating errors) and the different levels at which software can be tested (e.g. unit vs. system testing).

We present directed random testing in the context of a particular testing activity: testing software components that export a collection of operations; examples include container data types and object-oriented libraries. The goal of directed random testing is to automate the creation of unit test cases that check the operations for expected behavior, and that can complement a suite of manually-generated test cases. This section discusses aspects of software testing important for better understanding this context.

Since the goal of directed random testing is to generate test cases that complement a test suite, we starting by discussing the desired qualities of a test suite, and factors to consider when assessing an automated test generation technique. To relate directed random testing with other approaches to automated test generation, we give an overview of two previous approaches to automated test generation, random and exhaustive testing. To give the reader a better idea of the domain that directed random testing targets, we also describe the process of unit testing, and the creation of unit tests.

2.1 Desired Qualities of a Test Suite

Test engineers spend a large portion of their time crafting test cases. When creating a set of test cases (i.e. a *test suite*) for a software artifact, test engineers aim for the suite to have qualities that make it effective in revealing errors. A test suite should *adequately* test the multiple possible behaviors of the software under test; it should be *efficient* in the amount of resources it requires to execute; its test scenarios should represent *legal* uses of the software; and when it reveals an error, it should give *sufficient* and *succinct* clues to help the engineer debug the software.

- **A test suite should be adequate.** A test suite should sufficiently test the software, covering important scenarios that are likely to arise in practice. An ideal test suite contains a failing test case if and only if there is a failure in the software under test. (As we discuss below, such a test suite is exceedingly difficult to achieve.)

- **A test suite should be efficient.** An efficient test suite executes in reasonable time, allowing the developer to run it more frequently and discover errors earlier in the development cycle. One major source of inefficiency in test suites is the presence of *redundant* test cases: a test case is redundant if all the errors it reveals are revealed by other test cases. Redundant test cases increase the cost of executing a test suite. They can also waste time by signaling multiple failures that, upon manual inspection, are all caused by a single error.

- **A test suite's test inputs should represent legal uses of the software.** A test suite should test the software using inputs that it is required to handle correctly. Illegal test inputs may cause the software to behave unexpectedly, but not point to actual errors.

- **A test suite's test cases should give sufficient and succinct clues for debugging.** A useful test suite contains test cases that not only reveal errors in the software under test, but also make the process of determining the location of the erroneous code easier. For example, a test case that reveals an error in a binary search routine by running the routine on a 1-element array makes it easier to discover the source of the error than a test case that reveals the *same* error using a 1000-element array. More generally, given two test cases that reveal the same error, a more useful test case is a smaller test case.

It can be difficult to make the above properties precise, much less achieve them fully. For example, creating a fully adequate test suite that has a failing test case exactly when the software contains an error would actually constitute proof of the program's correctness [95], and proving a program correct is known to be difficult. Thus, an ideal test suite is typically unachievable within reasonable time. Instead, test engineers aim to create a test suite that test as many possible behaviors, preferably behaviors that reveal *critical* errors rendering the system unusable or open to security attacks. For example, when developing some of its products, Microsoft uses a four-point severity scale when rating errors [70], with "1" for a fatal error that renders the system unusable (and is always fixed prior to releasing the software, if discovered in time) and "4" a minor error that may cause inconvenience but does not affect core usability (and may never be fixed).

The other desired qualities can be equally difficult to achieve in practice. Detecting redundant test cases can be challenging in the face of software modifications: a test case that may be redundant in one version of the software may be non-redundant in a different version. Similarly, creating succinct test cases is challenging because extraneous test data for one version of the software may be crucial to revealing an error in a different version. Finally, creating legal inputs to the software can be challenging because what constitutes a legal input is often not written down precisely.

2.2 Automated Test Generation

The complexity of modern software means that the number of possible execution scenarios is extremely large. Manual test generation efforts often miss many important scenarios; the resources available to a development team (most importantly, human time) are usually not sufficient to cover all cases. Techniques that automate the creation of new test cases can help an engineer discover scenarios that escape manual testing, decreasing the cost of testing and increasing the quality of the software under test.

A fundamental question to ask when assessing a new automated test generation technique is whether a test engineer equipped with the new technique can be more productive in creating a test suite with the qualities described in Section 2.1, in a scalable and cost-effective manner.

- **Scalability.** The scalability of a technique refers to the technique's ability to give useful results even as the software under test grows in size or complexity. For example, a technique may fail to scale if its algorithms take too long or are unable to analyze language features present in the code.

- **Cost effectiveness.** For an automated test generation tool to succeed in a real industrial environment, it must reveal errors important enough to be fixed, and it must do so more economically than the engineer could do so without the technique. A technique may fail to be cost effective if it requires the user to spend manual effort disproportionate with the utility of the results.

The next three sections describe different approaches to automated test generation and discusses their scalability, cost effectiveness, and ability in helping an engineer create test cases. Sections 2.2.1 and 2.2.2 describe two established approaches to automated test generation, random testing and exhaustive testing. Each section describes the distinguishing features of each approach, and discusses the strengths and weaknesses that these features impart on techniques that implement the approach. Section 2.2.3 describes directed random testing, a new approach that combines elements from random and exhaustive testing.

2.2.1 Random Testing

Random testing generates test inputs randomly from the input space of the software under test [46]. Researchers and practitioners have implemented random test tools for many domains, including Unix utilities [72], Windows GUI applications [34], Haskell programs [20], and object-oriented programs [22, 82, 19].

The fundamental feature of a random test generation technique is that it generates test inputs at random from a grammar or some other formal artifact describing the input space. For example, "fuzz testers" [72] are tools that generate random sequences of bits or characters that can be used as input to stream-processing programs (e.g. Unix utilities). Random testers for file systems generate sequences of file system operations from a predetermined grammar of operations [44]. Random test generators for object-oriented classes generate sequences of method calls from a grammar of valid methods calls [22, 82, 19]. The distribution that a random tester uses when generating inputs need not be uniform. Some generators allow the user to specify distributions for selecting inputs or limit the domain of values [20]. The key is that when a choice is to be made in constructing a test input, a random generator makes the choice randomly, not deterministically.

Strengths of Random Testing

Random testing is an attractive test generation technique for two principal reasons.

- **Random testing reveals errors.** Many studies show that random testing is effective in creating test cases that reveal errors not found by other methods. For example, in a recent study researchers at NASA applied a random testing tool to a file system used aboard the Mars rover [44]. The generator created hundreds of failing tests that revealed previously unknown errors, despite the fact that many other testing techniques (manual and automated) had already been applied to the system. Section 11 describes many other studies that show random testing to be effective in creating error-revealing test cases.

- **Random testing is cost effective.** Random testing tools are easy to implement, and require little or no up-front manual effort on the part of the user. Together with their ability to reveal errors, this makes random testing attractive from a cost-benefit standpoint: a random testing tool can quickly generate a large volume of inputs that explore a large number of behaviors, regardless of the size or internal implementation of the software under test. For example, a fuzz tester that checks for abnormal program termination given random input can be applied to a stream-processing utility, a media player, or a web browser, with little or no modification.

Pitfalls of Random Testing

Random testing tools can reveal errors in a cost effective manner. However, they also suffer from deficiencies that can negatively impact their effectiveness.

- **Random testing can create many illegal inputs.** An illegal input is an input that violates the requirements expected by the program or unit under test. For example, a binary search procedure expects its input to be sorted, so most inputs generated at random would be illegal. Most random test generator generate inputs from a grammar that is easy to infer mechanically or hard-code in the tool, but may produce illegal inputs. For example, fuzz testers create random sequences of characters, but some Unix utilities expect input files with specific formats; for such utilities, a fuzz tester will generate many illegal inputs.

- **Random testing can create many equivalent inputs.** Random testing can produce many equivalent inputs that exercise the same behavior of the software, or that are indistinguishable for the purpose of revealing a certain kind of error. For example, when testing a binary search routine, only the ordering relationship among the elements in the array is important, not the actual values of the elements. Two arrays containing different elements, but the same ordering relationship among the elements, are equivalent. Most random testing tools do not attempt to detect equivalent inputs, and can easily generate a large number of them, wasting time by repeatedly exercising the same behaviors in the software under test.

- **Random testing creates long test cases.** Random testing tools usually produce test cases with large inputs (e.g. large streams of characters, large files, or long sequences of operations). Studies suggest that random testing is in fact more effective in revealing errors when it generates long tests, even when such errors can be revealed with shorter test cases [28, 6]. Unfortunately, large inputs can decrease the productivity of the engineer who may have to inspect them. A simple, concise test case can help an engineer quickly discover the cause of a failure. A long and complex test case can obfuscate the cause of the failure and increase the amount of time spent debugging.

2.2.2 Exhaustive Testing

Exhaustive testing techniques generate all inputs up to some input size bound. Two examples of exhaustive testing techniques are model finding (e.g. [60]) and model checking [21]. Some exhaustive generation techniques enumerate all inputs up to a given size [96, 4, 87], while others derive a set of constraints describing each non-equivalent input, and attempt to solve the constraints to yield concrete inputs [62]. The fundamental feature distinguishing exhaustive testing from random testing is that an exhaustive testing technique generates all inputs within a finite bound, typically in a deterministic fashion. Even for small bounds, the number of inputs can be infeasible to generate within reasonable resources (time or space). To combat this problem, exhaustive generation tools usually incorporate techniques to avoid generating illegal or equivalent elements from the input space. To avoid generating illegal inputs, some tools require invariants (constraints that specify expected properties of inputs) and restrict generation to inputs that satisfy the invariants. To avoid generating equivalent (or symmetric) inputs, exhaustive testing techniques may analyze the internal implementation of the software under test, to determine when two inputs might be indistinguishable to the software.

Strengths of Exhaustive Testing

- **Exhaustive testing guarantees coverage.** Because it tests the software on all small inputs, exhaustive testing guarantees complete coverage of the input space, up to the given bound. Some proponents of this approach argue, and our own experience supports, that most errors can be revealed by small test inputs [54, 4]. This suggests that testing a program exhaustively on small inputs is an effective way of revealing many of the errors present.

- **Exhaustive testing avoids generating useless inputs.** By pruning the input space, exhaustive testing techniques reap two benefits: they do not waste computer time exploring useless portions of the inputs space, and they do not waste human time because the engineer does not have to inspect false error reports resulting from illegal inputs, or multiple failing tests resulting from equivalent inputs.

Pitfalls of Exhaustive Testing

Exhaustive testing can suffer from scalability problems due to both the size of the input space and the complexity of the software under test. It can also be expensive in terms of human time required to write invariants, abstractions, and other artifacts necessary to prune the input space.

- **Exhaustive testing suffers from lack of scalability.** Exhaustive testing can fail to scale due to the size of input space or the complexity of the software under test. For many software artifacts, the size of the input space is too large even for relatively small input bounds, and even using techniques that prune the input space. Techniques that analyze the code under test can fail to scale if the code is too complex or uses features not handled by the analysis. For example, in a previous study, we found that a symbolic-execution-based technique was able to generate inputs for only 10% of the software artifacts used in the study due to limitations of constraint solving and to code that used features of the programming language not handled by the symbolic execution engine [26].

- **Exhaustive testing can require a high level of manual effort.** Most exhaustive techniques require the user to create artifacts like specifications, drivers, or abstractions. This can make exhaustive techniques costly to use and less likely to be adopted. In a recent study we conducted at Microsoft [81], we discovered that a team of test engineers was more receptive to using a directed random test generator than an exhaustive test generator that could in principle create a more comprehensive set of tests. The directed random test generator required no up-front effort, and the exhaustive generator required the engineers to write parameterized tests [89]. The team found errors using the random generator because they were able to immediately run it on their code base. They found no errors with the exhaustive generator because, not knowing if their manual effort would be cost effective, they wrote few parameterized tests.

2.2.3 A New Approach: Directed Random Testing

Directed random testing combines features from random and exhaustive testing. In particular, it combines random generation of inputs with pruning techniques that detect and discard illegal or equivalent inputs. Previous to our work, pruning techniques had only been applied in the context of non-random approaches [96, 4, 87].

Like random testing, directed random testing randomizes the generation of test inputs. Directed random testing generates new inputs incrementally by combining previously-generated inputs with new operations. (The approach is applicable to inputs that can be created via recursive applications of operations.) The generator maintains a *component set* consisting of previously-created inputs. To create a new input, the generator combines randomly-selected inputs from the component set, using them as arguments to new operations.

Like exhaustive testing, directed random testing *prunes* the input space. It does this by discarding an input from the component set if it is illegal or equivalent to an input already in the pool. The pruning is heuristic: the generator executes the software under test on a new input and determines if it is illegal or equivalent to another input based on the software's runtime behavior.

Strengths and Pitfalls of Directed Random Testing

By randomizing the generation of inputs, directed random testing aims to preserve random testing's scalability and effectiveness in revealing errors. Directed random testing is also likely to share some of random testing's pitfalls, such as its tendency to generate long test cases that are difficult to understand, and a lack of coverage guarantee. By heuristically discarding inputs that appear illegal and equivalent inputs, the approach aims to benefit from a strength of systematic testing, namely the pruning of large portions of the input space. To avoid systematic testing's weakness—potentially-intensive manual effort in specifying preconditions, abstractions, etc.—directed random testing uses heuristics (based on observing the runtime behavior of the software under test) to determine if an input is legal or illegal, and new or redundant. In later chapters, we describe experiments and case studies shed more insight into the strengths and weaknesses of the approach, and the effectiveness of the heuristics.

We develop the directed random testing approach in the context of unit testing. A unit test case checks the behavior of a single component or a group of related components in the software under test. To better understand the goal of the technique, Section 2.3 discusses the purpose of unit testing and the process of creating unit test cases.

2.3 Unit Testing

Most software applications are structured as collections of components, each implementing part of the functionality of the application. Testing an application as a whole is known as *system testing*, and testing each component individually is known as *unit testing*. For example, consider a command-line Java application (call it "polyplot") that takes as input a polynomial and outputs a file containing a plot of the polynomial. The polynomial, plot interval, and plot type are specified as command-line arguments, for example:

```
> java Polyplot --interval=1:10 -output=myplot.pdf (1/2)x^2+x
```

internal representation
of polynomial classes

```
public class Rat {
  // numerator
  public int num;
  // denominator
  public int den;
  // approximation
  public double approx;
}

public class Mono {
    // coefficient
   public Rat coeff;
   // exponent
   public int exp;
}

public class Poly {

 // Array containing
 // monomials elements.
 public Mono[] ms;
 ...
}
```

method `Poly.add`
(contains an error)

```
1. public Poly add(Mono m) {
2.
3.  // Will contain new monomial elements.
4.  List<Mono> mList=new ArrayList<Mono>();
5.
6.  // Go through the elements of this
7.  // polynomial, inserting the monomial m
8.  // in the appropriate place in mList.
9.
10. for (int i=0;i<ms.length;i++) {
11.   if (ms[i].exp == m.exp) {
12.    Rat sum = ms[i].coeff.add(m.coeff);
13.    Mono msum = new Mono(sum, m.exp);
14.    mList.add(msum); // ERROR
15.    continue;
16.   }
17.   if (ms[i].exp < m.exp) {
18.     mList.add(m);
19.   }
20.   mList.add(ms[i]);
21. }
22. return
23.  new Poly(mList.toArray(new Mono[0]));
24. }
```

Figure 2-1: Left: fields of the polynomial classes. Right: the `add` method (defined in class `Poly`) that adds a polynomial and a monomial and returns the result. Method `add` contains an error: it should only add the monomial `msum` to the list `mList` if the monomial has a non-zero coefficient, but it unconditionally adds the monomial (line 14), leading in some cases to a violation of `Poly`'s representation invariant. See Appendix A for a full code listing.

A system test case for polyplot would call the program on a command-line string like the one above, and check that the correct plot was created. On the other hand, a unit test for polyplot might check that a single component inside polyplot behaves correctly; for example, that the Java method `Poly.add` that adds polynomials (Figure 2-1) functions correctly.

System and unit testing are complementary approaches to testing. System tests can reveal errors in the end-to-end behavior of the application, while unit tests can reveal errors in individual components earlier, before the components are assembled into a larger system. The techniques presented in the thesis generate test cases at the unit level.

Figure 2-2 shows a unit test case `testAdd` for method `Poly.add`. It checks that the object returned by `add` satisfies the representation invariant [66] of class `Poly`, shown in Figure 2-1. A representation invariant is a logical property specifying the valid internal states of an object. A public methods like `add` should preserve the

Unit test for `Poly.add` method

```
1.  public static void testAdd() {
2.
3.  // Input creation code.
4.  Poly p =
5.  new Poly(new Mono(new Rat(2,1),3));
6.  Mono m  = new Mono(new Rat(-2,1),3);
7.
8.   // Test call.
9.   Poly p2 = p.add(m);
10.
11.  // Test oracle.
12.  boolean inv = p2.checkRep();
13.  Assert.assertTrue(inv);
14. }
```

`repOk` method in class `Poly`.

```
public class Poly {

 // Checks representation invariant:
 // 1. Elements sorted in order of
 //    decreasing exponent.
 // 2. No two monomials have the same
 //    exponent.
 // 3. No monomial has 0 coefficient.
 boolean repOk() {
   for (int i=0 ; i<ms.length ; i++) {
     if (i>0 && ms[i-1].exp<=ms[i].exp)
       return false;
     if (ms[i].coeff.num == 0)
       return false;
   }
   return true;
 }
}
```

Figure 2-2: Left: unit test case for method `add` in class `Poly`. The prep code consists of method calls that create input to the method under test. The call of the method under test is followed by a check that the object returned by `add` preserves the invariant of the class. Right: class `Poly` defines a method `repOk()` that checks the representation invariants of its argument, returning `false` if the invariant is violated. The unit test uses the `repOk()` method as its oracle.

representation invariants of their inputs and return objects.

The test case shown is written in JUnit [68], a format for writing unit tests where each test is written as a method in a test class. As the test case shows, a unit test case has three components. The *input creation code* (lines 4–6) creates the objects used as input to the call. The *test call* (line 9) calls the method under test. The *test oracle* (lines 12–13) checks the expected behavior of the method under test.

When executed, `testAdd` reveals an error in the implementation of method `add`. The method incorrectly adds a polynomial p and a monomial m, if m is the negation of an element in p (for instance, if $p = 2x^3 + 1$ and $m = -2x^3$). The error is in lines 12–15 (Figure 2-1, right), which handle the case where m has the same exponent as an element of p. Line 12 adds the coefficients of the monomial and the polynomial element. At this point, the code should check if the resulting coefficient is zero, indicating that the monomial and the polynomial element are equal; if they are, the code should proceed to the next element without adding the zero-coefficient monomial to the list `mList`. The code fails to do this, and adds the monomial unconditionally to `mList`, which can result in a return value containing a monomial with a zero coefficient. This in turn leads to the unit test failing (line 13 in `testAdd`, Figure 2-2). Note that the oracle for a test case need not check for every required property of a method under test to be useful. For example, `testAdd` does not check that `add` returns the correct polynomial, only one that respects its object invariant.

As the example shows, creating a unit test involves creating two artifacts: a test input for the component under test, and a test oracle to check that the component under test behaves as expected. This thesis addresses the input creation problem, and assumes the existence of a procedure, like method `Poly.repOk` in the example, that can check a correctness property of the component based on its execution. The upcoming chapters address the problem of generating inputs that achieve high behavioral coverage of the components under test and that reveal errors. The techniques create test inputs by chaining together builder operations to create values of the sort required as input to a component under test, much like `testAdd` chains together calls constructor calls of `Rat`, `Mono` and `Poly` (lines 4–6 in the test) to create the inputs to `add`.

We note here that chaining operations is not the only possible way of creating input values for a component. For example, another approach, exemplified by tools like TestEra [61] and Korat [16], is to create inputs by directly manipulating the internal representation of objects (e.g., creating a `Mono` object by setting the `coeff` and `exp` fields to particular values, instead of calling the constructor). The advantage of a representation-level approach like TestEra or Korat's is that it does not require an existing set of operations to create input values, and it allows for direct creation of input values that may be difficult (or even impossible) to achieve using a given set of operations. The main disadvantage is that it typically requires a significant amount of manual effort. Because arbitrarily setting fields results in many invalid values (e.g. `Rat` objects with zero denominator), the user needs to specify constraints that specify the values that constitute valid assignments to the fields. Naïvely chaining together operations can also result in many illegal values. However, one of the key insights in directed random testing is that the number of illegal (and redundant) values can be reduced if the chaining of operations is enhanced by executing the operations and using the result of the execution to heuristically determine if a sequence of operations represents a legal input. This works because operations leading to illegal inputs often result in unexpected behavior. For example, calling `Rat`'s constructor with a `0` denominator results in an `IllegalArgumentException`. A field-oriented approach could also benefit from runtime information, by executing operations on generated values and observing the result of the execution to learn if the values are illegal. However, a field-oriented approach may not be able to leverage operations to create *legal* values; for example, by calling a 0-argument constructor for a class, an operation-based approach may easily obtain a legal object for the class that may be difficult to create with a field-based approach. Also, a field-oriented approach does not benefit form the pruning that an incremental, operation-based approach achieves by discarding illegal or redundant inputs from the generation process.

Chapter 3

Generating Inputs with Terms

This chapter describes how to generate test inputs by viewing operations as functions, and generating terms over those functions. Terms are useful because they provide a language for constructing input values: we can construct a new test input by generating a new term, evaluating the term, and using the value resulting from the evaluation as a test input.

After formally describing the syntax and interpretation of terms, we describe two strategies for mechanically generating terms. *Top-down generation* creates terms recursively, starting from the outermost operation and recursively creating arguments for the operation. *Bottom-up generation* begins by creating *constant terms* that require no arguments, and combines smaller terms to yield larger ones. Each approach has unique advantages; the advantage of bottom-up generation, which is the basis of directed random testing, is that it enables greater control in the generation process, as we explain at the end of the chapter.

3.1 Signatures

Terms are composed of recursive operation applications, with each operation taking a collection of input terms and producing a larger term. Before describing terms formally, we describe *signatures*, which describe the kinds of inputs operations take, and the kind of output they produce. (The concepts of signatures and terms are not our creation; they are a formalism used often in computer science, for example in the data type specification literature [67].) To motivate the definitions, recall the operation `Mono(Rat,int)` that creates a monomial. The operation takes two inputs: a `Mono` value, and an `int` value, and produces as output another `Mono` value. A signature describes the input/output constraints of an operation. Signatures are used by a term generator to create meaningful terms with consistent input and outputs.

An *operation signature* is a tuple of symbols $\langle opname, r_1, \ldots, r_k, r_{out}\rangle$. We refer to the first element as the *name* of the operation, to the elements $r_1, \ldots, r_k$ as the *arguments* of the operation, and to r_{out} as the *result* of the operation. We refer to $r_1, \ldots, r_k$, and r_{out} as *sorts*. (Sorts correspond to "types" in programming languages.) For better readability, we typically write an operation $\langle opname, r_1, \ldots, r_k, r_{out}\rangle$ as "*opname* : $r_1 \times \ldots \times r_k \rightarrow r_{out}$." As an example, the following set of operation signatures describes the operations from Figure 1-1:

$C = \{$ `Poly` : `Mono[]` $\rightarrow$ `Poly`, `Mono` : `Rat` $\times$ `int` $\rightarrow$ `Mono`, `Rat` : `in` $\times$ `int` $\rightarrow$ `Rat` $\}$.

Some additional terminology:

- We use the terms *operation signature* and *operation* interchangeably when there is no potential for confusion.
- Given an operation *op*, we denote its name as *op.name*, its arguments as *op.args* and its result as *op.result*.
- We say an operation *op* is *k-ary* if $|op.args| = k$.

3.2 Terms

A term is a tree that represents a value obtained by the recursive application of operations; it yields an input value when the operations are evaluated. For example, the following tree (written serially, with parentheses to show the structure) represents the construction of the polynomial value $2x^3+1$ using the polynomial operations from Figure 1-1:

`Poly(Mono(Rat(2,1),3),Mono(Rat(1,1),0))`.

We define the set of *terms* over a set of signatures *ops* inductively, as the smallest set satisfying the following properties.

1. If *op* is a 0-ary operation, then $\langle op \rangle$ is a term.
2. If $op = opname : r_1 \times \ldots \times r_k \rightarrow r_{out}$ for $k > 0$, and for $1 \leq i \leq k$, t_i is a term whose outermost operation has result sort r_i, then $\langle op, t_1, \ldots, t_k \rangle$ is a term.

Term-related terminology:

- We write *terms*(*ops*) to denote the set of all terms over an operation set *ops*.
- We call the terms defined via rule 1 *constant terms*, and denote them *constants*(*ops*). For example, the term `Poly()` is a constant term. We treat integer literals as constant terms. For example, we associate with the literal `1` a 0-ary operation `0` :$\rightarrow$ `int`, and represent the literal as the term $\langle$`0` :$\rightarrow$ `int`, $\langle\rangle\rangle$. (Below we may write a constant term like $\langle$`0` :$\rightarrow$ `int`, $\langle\rangle\rangle$ informally, and more briefly, as "0.")
- We say a term is an *r-term* if its outermost operation has result sort *r*.

For ease of reading, when writing a term in the text, we use the parenthesized syntax with typewriter font we have used informally so far (e.g. `Rat(1,1)`), not the more verbose tuple representation (e.g. $\langle$`Rat` : `int` $\times$ `int` $\rightarrow$ `Rat`, `1`, `2`$\rangle$).

3.3 Interpreting Terms

The recursive structure of terms reflects the compositional creation of values using operations. For example, to create a `Poly` value, we must first create a number of `Mono` values. These, in turn, require creating some `Rat` values. The term `Poly(Mono(Rat(2,1),3), Mono(Rat(1,1),0))` expresses the construction process that yields $2x^3 + 1$. However, terms alone are syntactic constructs that have no meaning: they are only valuable because of their connection to runtime values. To formalize the connection between terms and values, we assume the existence of a universe of values *Vals*, and associate with each sort r a set V_r of values (of the given sort). This is a straightforward analogy to the connection between sorts and values in real code. For example, associated with each sort `Rat`, `Mono` and `Poly` is a set of values corresponding to the runtime values corresponding to rational, monomial and polynomial objects.

We view operations as mappings that generate values from other values. With each operation $op = opname : r_1 \times \ldots \times r_k \rightarrow r_{out}$, we associate a mapping $M_{op} : V_{r_1} \times \ldots \times V_{r_k} \rightarrow V_{r_{out}}$. Given a term t, the *value* for t, denoted $[\![t]\!]$, is defined as a recursive application of mappings based on the structure of t, in the obvious way. We may also say that t *evaluates to* or *produces* $[\![t]\!]$. The syntactic structure of terms along with the functional interpretation we just gave is known as a many-sorted algebra, a basic concept in data type specifications [67].

Given a set of operations, there may be values in the universe that cannot be created by a recursive application of operations, and will thus be outside the scope of any technique that creates values by generating terms over the set of operations. For example, given an operation set consisting of the operations from our polynomial example and constants { `-10`, ..., `10` }, the polynomial x^{11} will be outside the scope of any term-based generator, since polynomial multiplication is not defined. (The real classes on which the example is based do provide multiplication.) This is a fundamental limitation of any approach that uses operations to create values, and it can prevent a term-based generator from revealing errors. In our experience, the space of values achievable via operations is typically very large, and techniques that explore within this space are effective in revealing many errors in the software artifacts under test. Another issue is that the operations under test for a given software artifact may not behave like functions, which we assume in our interpretation. For example, in Java, calling the `Rat` constructor twice on identical inputs (e.g. `Rat(2,1)`) returns two different objects. For now, we ignore this, and assume that operations can be interpreted as functions; this makes it easier to describe the key ideas in our approach. In Chapter 5, we deal with problems with our interpretation of operations, and propose a more general interpretation that handles non-determinism and side effects.

3.3.1 Illegal Input Values

An operation may impose additional requirements on its inputs not captured by the operation's argument sorts. For example, Figure 1-1's comments indicate that the operation `Poly(Mono... elements)` requires that the `elements` array be not `null` and be sorted in order of decreasing exponent. Validity constraints on inputs are also known as *preconditions*. Preconditions limit the set of values that constitute legal input to an operation beyond the input's sort.

To account for the presence of preconditions, we associate with each operation $op = opname : r_1 \times \ldots \times r_k \rightarrow r_{out}$, a set of k-tuples

$$LI = \{\langle v_1, \ldots, v_k \rangle : v_i \in V_{r_i} \text{ for } 1 \leq i \leq k\}$$

representing the input value tuples that are legal inputs for each argument of the operation. We call this set the *legal input set* of the operation. We say that a term is a *legal term* if the inputs to every operation evaluate to values within the legal input set of the operation.

A term generation technique that avoids illegal terms can be more effective in revealing errors for two reasons. First, executing a component under test with illegal input values often causes early termination of the component. For example, calling method `Poly.add(Mono)` with a `null` argument results in a `NullPointerException` when the second statement (`for (int i=0;i<ms.length...)` is executed for the first time (Figure 2-1). Inputs that result in early termination fail to properly exercise code deeper in the component under test, and can result in low behavioral coverage. Second, calling a component under test with illegal inputs can result in arbitrary behavior that may appear to (but does not) reveal an error. For example, a test case that reports a potential error because `add` results in a `NullPointerException` indicates a potential error in the method, until the culprit is discovered to be a `null` input value. Spurious error reports cause a test engineer to spend time inspecting false alarms.

3.3.2 Error Values

The goal of the techniques presented in the thesis is to generate test inputs that reveal errors. We model errors by identifying some values in the universe as *error values*. To identify error values, our techniques require as input an *oracle*: a predicate on values $O(v)$ that returns *false* when a given value is an error value. We say that a term t is *error-revealing* if $O([\![t]\!]) = \textit{false}$. (Our use of the term "oracle" is slightly more restricted than other uses in the literature [52] that define an oracle as a predicate $O(v, f(v))$ that returns *true* if the result $f(v)$ of an operation on a value v is as expected.)

We already saw an example of an oracle: Figure 2-2 showed method `Invariant-Checker.check(Poly)` that returns `true` when the given polynomial value satisfies the object invariant for class `Poly`. A component under test (like `Poly.add`) that returns an error value (given legal inputs) is likely to contain an error in its implementation.

When developing techniques that generate terms, we assume the testing goal is to ensure proper behavior of an operation *that is given valid inputs.* In some cases, it makes sense to test software on illegal inputs, to check for incorrect or missing validity

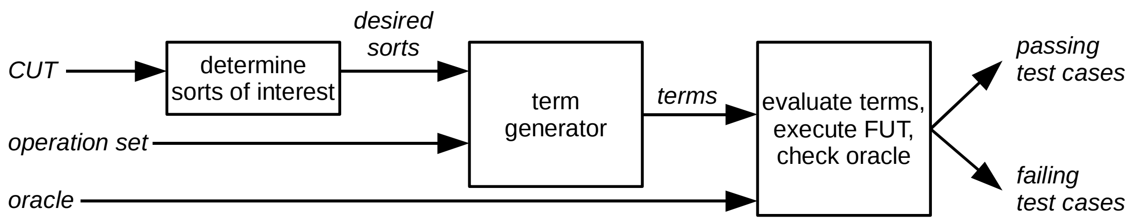

Figure 3-1: A technique that creates test cases for a component under test (CUT) by generating terms. The technique takes as input the CUT, a set of operation used to create terms, and an oracle to determine if the CUT returns an error value. The technique outputs a set of passing test cases (tests that pass the oracle) and failing test cases (that fail the oracle).

checks in the software under test. (This testing activity is sometimes called “robustness testing.”) Robustness testing can be useful because many software artifacts are supposed to check the legality of their inputs and fail gracefully if preconditions are violated. For example, this is usually the case in the core utility libraries for languages like Java and .NET. In this case, a failure of the software resulting from illegal operation applications may reveal a legitimate error in the legality-checking code. Our techniques can be used for robustness testing, by assuming there are no preconditions.

3.4 Generation Strategies for Terms

This section presents two basic strategies for generating terms over a set of operations. The strategies are examples of *undirected* random testing: they generate terms purely based on the signatures of the operations. In later chapters, we explore *directed* random testing techniques that use additional information to direct the generation away from illegal and redundant inputs.

The term generators we present can be used as part of a simple test case generation technique, shown in Figure 3-1. The technique takes as input a component under test (CUT), a set of operations, and an oracle that determines error values. The technique first extracts the *desired sorts*, meaning the input sorts to the component under test. Next, it uses a term generator (which takes as input a desired sort and the set of operations) to generate terms of the desired sorts. Finally, the technique evaluates the terms to obtain input values, executes the component under test on the input values, and checks the oracle after each execution. Terms that cause oracle failures are output as *failing test cases*, and other terms are output as *passing test cases*. An example of a test case that may be generated by this process is the unit test case from Chapter 2 (Figure 2-2).

3.4.1 Top-Down Generation

Terms are defined inductively, so a natural approach to generating them is to use recursion. A top-down generator creates terms recursively. To create a term of a

```
Procedure Td
Input: Operation set $ops$, desired sort $r$, time limit $t$, maximum depth $d_{max} \geq 0$.
Result: A set of $r$-terms.
1 $terms \leftarrow \{\}$
2 while time limit $t$ not expired do
3     $t \leftarrow$ Gen($ops, r, 0, d_{max}$)
4     $terms \leftarrow terms \cup \{t\}$
5 return $terms$
```

```
Procedure Gen
Input: Operation set $ops$, sort $r$, depth $d$, maximum depth $d_{max} \geq 0$.
Result: An $r$-term.
1 if $d = d_{max}$ then
2     return rchoose $\{t \in constants(ops) : t$ is an $r$-term$\}$
3 $op \leftarrow$ rchoose $\{op \in ops : op.result = r\}$
4 let $opname : r_1 \times \ldots \times r_k \rightarrow r$ be $op$
5 return $\langle op$, Gen($ops, r_1, d+1, d_{mas}$), ..., Gen($ops, r_k, d+1, d_{max}$)$\rangle$
```

Figure 3-2: A top-down random generator for terms. The operator **rchoose** selects one element uniformly at random from a set.

desired sort, a top-down generator selects an operation, and recursively generates terms to use as arguments to the operation. Figure 3-2 shows a procedure, Td ("top-down"), that implements a random version of the strategy. Td takes as input a set of operations *ops*, a desired sort r, a time limit t, and a maximum depth d_{max} that bounds the recursion. Td returns a set of r-terms, containing as many terms as the procedure is able to generate within the specified time limit.

To generate a term, Td repeatedly calls sub-procedure Gen (which generates a single r-term), and adds the new term to the term set that is eventually returned. Procedure Gen takes a depth parameter d as input (Td passes 0 as the initial depth). If d is equal to the depth limit d_{max} (the base case), Gen returns a constant term, selected uniformly at random from all constant terms. We assume that for every sort r in *ops*, there is at least one constant r-term in *terms*(*ops*). Otherwise, if $d < d_{max}$, Gen selects an operation with result sort r (again uniformly at random), and recursively calls itself to generate an r_i term for every argument i of the operation.

To illustrate how Td works, suppose that Td calls Gen with (1) the polynomial operations, augmented with operations yielding primitive terms like `1` and `2`, (2) desired sort `Rat`, (3) depth 0, and (4) maximum depth 10. Since $d < d_{max}$, Gen selects an operation at random from any of the operations with return sort `Rat`. There are two possibilities, `Rat(int,int)` and `Rat.add(Rat)`. Suppose Gen selects the `add` operation. In that case, it will recursively invoke itself twice, to generate two `Rat` terms. Consider the first recursive call, and suppose this time Gen selects

`Rat(int,int)`. Two more recursive calls of GEN will yield integer terms (say, `2` and `3`), yielding the term `Rat(2,3)`. Now, consider the second recursive call, and suppose it also selects `Rat(int,int)` and yields `Rat(1,2)`. The final term will thus be `add(Rat(2,3),Rat(1,2)`. Note that the depth limit simply prevents an infinite recursion in GEN, but it does not prevent TD from generating terms of depth less than d_{max} because the recursion can always terminate if a constant operation is selected. For example, the depth of the example term generated is 3.

TD is only one of several possible ways of implementing a top-down strategy. For example, a different implementation might use a different distribution when selecting operations, or even replace some or all random choices (lines 2 and 3 in GEN) with loops that iterate through all the possible choices. The common characteristic of a top-down implementation is that a term is generated recursively, starting with a desired sort r, and creating sub-terms that provide inputs to an operation that returns an r value.

JCrasher [22], a popular term-generation tool, is an implementation of top-down random generation. It generates terms in a manner similar to TD. Chapter 8 evaluates JCrasher against other generation approaches, including directed random testing.

3.4.2 Bottom-Up Generation

A top-down generator creates terms recursively. A bottom-up generation strategy involves instead generating terms incrementally, combining previously-generated terms with new operations to yield larger ones. To describe the strategy, it is useful to define the set of all terms that are achievable by combining other terms. The *image* of a set of terms C with respect to a set of operations *ops*, denoted $I(C, ops)$, is the set of terms whose sub-terms are elements of C:

$$I(C, ops) = \{t \in terms(ops) : t = n(t_1, \ldots, t_k) \text{ and } t_i \in C \text{ for } 1 \le i \le k\}$$

A bottom-up generator maintains a set of terms called a *component set,* initially empty. It generates new terms by iteratively augmenting the component set with elements from its image. Figure 3-3 shows a procedure, BU ("bottom-up"), that implements the bottom-up strategy. BU takes as input an operation set *ops*, a desired sort r, and a time limit t. (Because BU does not use recursion, it needs no recursion limit parameter.) Next, BU repeatedly performs a sequence of two steps, until the time limit expires. The first step selects a term from the image of the component set at random, and the second step adds the term to the component set. When the time limit expires, BU returns the set of r-terms in the component set. To illustrate the procedure, we consider the first few iterations of a sample run of BU, using the polynomial operations. In the first step, the component set is empty, so only constant terms are part of the image. Suppose BU first selects the constant term `Poly()` and adds it to the component set. The image of the component set in the second iteration is the same as that in the first iteration, because no new terms can be generated using `Poly()` alone. Suppose that term "`1`" is selected next and added to the component set. In the third iteration, the image of the component set contains

```
Procedure Bu
Input: Operation set $ops$, desired sort $r$, time limit $t$.
Result: A set of $r$-terms.
1 $C \leftarrow \{\}$
2 while time limit $t$ not expired do
3     $t \leftarrow$ rchoose $I(C, ops)$
4     $C \leftarrow C \cup \{t\}$
5 return $\{ t \in C : t$ is an $r$-term $\}$
```

Figure 3-3: A bottom-up random term generator. The generator maintains a component set of terms C, which it repeatedly augments with elements from C's term image.

term `Rat(1,1)`. Suppose that term is selected and added to the component set. In the fourth iteration, the image contains all the terms in the previous image, plus two new terms `add(Rat(1,1),Rat(1,1))` and `Mono(Rat(1,1),1)`. Notice that as new terms are added to the image, the set of possible new terms increases.

As with Td, other ways of implementing random bottom-up generation are possible. For example, the procedure could at each step augment C with several random terms in the image, instead of a single term, or even augment it with the entire image at each step (yielding a deterministic algorithm).

3.4.3 Random Generation: Top-Down vs. Bottom-Up

The top-down and bottom-up approaches to generation represent two points in the design space of term generation approaches. Each approach has relative advantages. We begin by discussing two advantages that top-down generation enjoys compared to bottom-up generation.

Top-Down Generation Produces More Diverse Terms

Because it generates every new term from scratch, a top-down generator can generate a more diverse set of terms than a bottom-up generator. The latter generates terms by composing previously-generated terms, and this can result in many terms with common sub-terms. To illustrate this phenomenon, Figure 3-4 shows the first ten terms generated in two actual runs of Td and Bu for the polynomial data types. Both runs use `Poly` as the desired sort, use arbitrary random seeds, and augment the polynomial operations with operations representing integer values `-10` to `10`. Also, when creating a term for operation `Rat(Mono...)`, we implemented both procedures to randomly select between 1 and 4 monomials as input. The implementation of Bu differs from Figure 3-3 in that it initially seeds the component with constant terms corresponding to primitive values, thus making them immediately available to create new terms.

First ten terms generated in a run of Td

illegal	`Poly(Mono(Rat(-8,-2),-8),Mono(Rat(4,7),0))`
illegal	`Poly(Mono(Rat(1,1),7),Mono(Rat(4,0),5))`
legal	`Poly()`
illegal	`Poly(Mono(add(Rat(6,1),Rat(-10,-9)),0),Mono(sub(Rat(2,7), Rat(0,1)),-4),Mono(sub(Rat(1,2),Rat(1,4)),-4))`
illegal	`Poly(Mono(Rat(3,3),0),Mono(add(Rat(-4,2),Rat(1,7)),3),Mono(Rat(-7,-4),-1))`
legal	`Poly(Mono(sub(Rat(1,2),Rat(4,3)),5))`
legal	`Poly()`
legal	`Poly()`
legal	`Poly(Mono(Rat(-5,2),7))`
illegal	`Poly(Mono(add(Rat(-9,-9),Rat(5, -6)),-5))`

First ten terms generated in a run of Bu

legal	`Poly()`
legal	`Rat(-9,9)`
legal	`Rat(7,4)`
legal	`Rat(-1, -7)`
legal	`add(Rat(-9,9),Rat(7,4))`
legal	`Mono(Rat(-1, -7),9)`
illegal	`Rat(4,0)`
legal	`Poly(Mono(Rat(-1, -7),9))`
illegal	`add(Rat(7,4),Rat(4,0))`
legal	`Poly(Mono(Rat(-1, -7),9),Mono(Rat(-1, -7),9))`

Figure 3-4: First ten terms generated by a run of algorithms Td and Bu, using an arbitrarily-chosen seed. The label next to each term indicates if it is legal or illegal with respect to the preconditions specified in Figure 1-1.

As the figure shows, many of the terms that Bu generates have common sub-terms. For instance, the term `Rat(-1,-7)` appears five times, and the `Mono` term `Mono(Rat(-1,-7),9)` appears four times. Td, on the other hand, generates every term randomly; the first 10 terms generated by Td contain 13 distinct monomial terms. The repetitive nature of Bu-generated terms means that a technique using Bu may waste testing resources by repeatedly exercising identical behavior in the software under test.

Top-Down Generation is More Goal-Directed

A second advantage of Td is that it is more goal-directed in generating terms for the desired sort. Each term generated in Td's main loop is guaranteed to produce a term for the sort of interest. On the other hand, Bu's incremental nature may make it impossible to generate n terms for a desired sort r in the first n iterations of its main loop, because operations producing r-terms may be disabled until terms of the appropriate input sorts are present in the component set. We can see this effect in

Figure 3-4 as well. In ten iterations through its main loop, Td generates ten `Poly` terms (8 syntactically distinct ones), while Bu generates only 3.

This advantage of Td over Bu is important when the goal is to generate inputs of a specific sort like `Poly`, as in the examples we have seen so far. However, the techniques we present in the following chapters have as their goal testing the operations themselves. In other words, we view operations as both building blocks for input values, *and* as the components under test. In this context, the advantage of Td over Bu disappears. For example, suppose that the testing goal is not to test the single component `Poly.add(Mono)`, but to test *all* the components (i.e. operations) declared in the `Rat`, `Mono` and `Poly` data types. Thus, we think of terms as both a means to build polynomial, monomial and rational input values, and as test inputs that exercise the operations they contain. In this case, all ten terms generated in the example run of Bu are useful test inputs. As a more realistic example, we have used Randoop (a tool based on bottom-up generation) to test the Java JDK library [40], and view the methods declared in the library both as operations used to create inputs, and as the components under test.

In the rest of the thesis, we assume that the testing goal is to test the set of operations given as input, not a single operation in particular. Thus, the procedures presented take as input a *set* of operations under test.

At this point, we have yet to explain why we base directed random testing on bottom-up generation. The following section explains our rationale.

Bottom-Up Generation Facilitates Pruning

The great advantage of bottom-up generation over top-down generation is that bottom-up generation has the crucial feature that *it generates new terms from old ones.* While not useful in a basic procedure like Bu, this feature is useful for directed random testing because it makes it possible to *influence* the generation process by restricting the terms allowed to be in the component set. This can be helpful in directing the generation process towards useful test inputs.

For example, notice that both Td and Bu generate a large number of illegal terms. In the runs from Figure 3-4, Td generates five (out of ten) illegal term, and Bu generates two (out of ten) illegal terms. (The fewer illegal terms of Bu are due to random luck; another random seed might have resulted in more.) As we discuss in Chapter 2, illegal inputs can be a problem with random testing techniques, and methods to avoid generating them could improve the effectiveness of random testing. The next chapter shows that, given a means of determining if a single term is illegal, a small modification to Bu that discards a term (and does not add it to the component set) if it is illegal yields a new bottom-up generation procedure that discards illegal terms like `Rat(4,0)` (generated in the run of Bu from Figure 3-4), but also *prunes* the term space, by preventing any term that has the illegal term `Rat(4,0)` as a sub-term, from being generated. The chapter also shows how a similar idea can result in pruning the space from redundant terms that lead to the same behavior in the software under test as terms already in the component set and can waste generation effort.

Chapter 4

Directed Random Testing for Terms

Directed random testing is a novel approach to test input generation. It extends bottom-up generation by discarding illegal terms (terms that the software under test is not required to handle) and redundant terms (terms that do not lead to the discovery of new errors) from the component set. The effect of discarding an illegal or redundant term term is not that a single term is removed from the space, but rather—due to the inductive nature of bottom-up generation—that useless portions of the term space are pruned, i.e. many useless terms are never generated.

This chapter presents directed random testing. The chapter introduces the different ideas making up directed random testing in phases, by presenting a series of techniques that add functionality to a basic bottom-up term generator. The first technique uses a user-supplied *legality predicate* that specifies what are legal input values for each operation, and uses the predicate to discard illegal terms from the component set. The second technique additionally uses an *equivalence relation* on terms, which it uses to discard a term if it is equivalent to one already in the component set. The concept of equivalence depends on the software artifact under test as well as the test oracle; we describe formally the desired property that an equivalence relation must satisfy, namely that discarding a term based on the relation does not prevent the generator from revealing an error. Using a legality predicate and an equivalence relation on terms, the bottom-up generator is able to prune illegal and redundant portions of the input space.

The second key idea behind directed random testing is the notion of *heuristic pruning*, which consists in *approximating* definitions of terms legality and equivalence based on the runtime behavior of the software under test. Heuristic pruning is important because it makes directed random testing a fully automated procedure, requiring only the software under test as input. One practical advantage of random testing tools is that they are typical fully automated, making them cost effective and easier to adopt.

```
Procedure Bu/L
Input: Operation set $ops$, time limit $t$, legality predicate $Leg$.
Result: A set of terms.
1 $C \leftarrow \{\}$
2 while time limit $t$ not expired do
3     $img \leftarrow \{ \langle op, t_1, \ldots, t_k \rangle \in I(C, ops) \ : Leg(op, t_1, \ldots, t_k) \}$
4     $t =$ rchoose $img$;
5     $C \leftarrow C \cup \{t\}$
6 return $C$
```

Figure 4-1: A bottom-up generator that avoids illegal terms.

4.1 Pruning Illegal Terms

Applying an operation to illegal inputs rarely constitutes a useful test. Illegal inputs usually lead to early termination or arbitrary but not error-revealing behavior. For example, evaluating the illegal term `Rat(1,0)` leads to an exception at runtime. The exception reveals no errors, but simply indicates that `Rat`'s precondition has been violated. It is wasteful for a test generation technique to generate illegal inputs: the time spent generating them could be better spent generating legal inputs that exercise the software under conditions that it is expected to handle correctly.

Figure 4-1 shows a procedure, Bu/L, that augments the basic bottom-up generation procedure from last chapter so that it avoids generating illegal terms. Bu/L requires as input a *legality predicate Leg* that determines if the inputs to a new term are illegal. Given an operation $op = opname : r_1 \times \ldots \times r_k \rightarrow r_{out}$ and terms $t_1, \ldots, t_k$ (of sort $r_1, \ldots, r_k$, respectively), $Leg(op, t_1, \ldots, t_k)$ should return *true* if $t_1, \ldots, t_k$ represent valid arguments to *op*. For example, $Leg(\texttt{Rat} : \texttt{int} \times \texttt{int} \rightarrow \texttt{Rat}, 1, 0)$ should return *false*, because the operation `Rat(int,int)` requires its second argument to be non-zero.

The procedure discards from the image any term that is illegal according to the legality predicate. This discarding not only prevents the selected term from being illegal, but it also *prevents* many illegal terms from ever being considered. For example, by discarding the illegal term `Rat(4,0)`, Bu/L prevents the future generation of illegal terms like `Mono(Rat(4,0),1)` or `add(Rat(7,4),Rat(4,0))` that use `Rat(4,0)` as a sub-term, because the sub-terms will never be in the component set. In other words, discarding a useless term early in the process can prune a large number of other useless terms later in the process. Notice that this kind of pruning is not possible in a top-down approach like Td that does not generate new terms from old ones. Other than filtering illegal terms after they were generated, it is not easy to prune illegal portions of the space with Td, because the algorithm maintains no information about previously-generated terms that it can use to direct the generation of new ones.

An *ideal* legality predicate *Leg* returns *true* exactly when the input terms represent legal input values to the operation. In other words, an ideal predicate is one that returns *true* on $Leg(op, t_1, \ldots, t_k)$ if and only if $\langle [\![t_1]\!] \ldots [\![t_k]\!] \rangle$ is in the operation's legal input set (see Section 3.3). A *non-ideal* legality predicate can deviate from the ideal in two ways. Fist, it can return *true* on an illegal term t. In this case, the component set may become populated with illegal terms (any term using t as a sub-term). Second, it can return *false* on a legal term. This will prevent Bu/L from exploring potentially useful portions of the term space. Creating an ideal legality predicate can be challenging, and in some cases, a predicate that is not ideal can also be effective. Section 4.4 describes a way to automate the creation of a non-ideal predicate that is effective in practice.

4.2 Pruning Equivalent Terms

The last section described a way to make bottom-up generation more effective by discarding illegal terms that are less likely to reveal errors. This section explores a second criterion for discarding terms, based on a concept of term *equivalence*, meant to capture the idea that for the purposes of creating error-revealing test inputs, it may be sufficient to explore only a subset of all terms, and that discarding new terms equivalent to already-generated terms can improve the error-revealing effectiveness of a bottom-up generator.

We begin our discussion of equivalence in Section 4.2.1 by introducing the concept in a simple setting, where the purpose is to discard useless terms generated by a term generation procedure, without altering the procedure itself. Next, we consider the more complex scenario of discarding equivalent terms *inside* a bottom-up term generator, which can result in pruning of the term space. We define *safety*, a desired property of an equivalence relation on terms. An equivalence relation is safe if discarding terms based on the equivalence relation does not cause a term generator to miss errors. Section 4.2.2 describes a method to establish safety for an equivalence relation, by decomposing the proof that a relation is safe into smaller proofs that separately link the equivalence relation with the oracle and each operation. Finally, Section 4.2.4 describes *efficiency*, an additional desired quality of an equivalence relation.

4.2.1 Safe Equivalence

The purpose of an equivalence relation on terms is to discard terms that do not contribute to the goal of discovering an error. For simplicity, we assume for now that there is a single error in the software under test revealed by an oracle O, thus it suffices to find one term t such that $O([\![t]\!]) = \textit{false}$.

To gain some intuition for the notion of equivalence, we begin by describing term equivalence, and the desired properties of an equivalence relation, in the simple context of filtering a collection of terms that have already been generated.

Equivalence for Filtering

Imagine that we are using a term generation procedure (e.g. TD, BU, or BU/L) that produces candidate terms, with the goal of discovering a term that reveals the error, i.e. a term t where $O([\![t]\!]) = \mathit{false}$. When a new candidate term is generated, we evaluate the term and check the oracle. Since there is a single error in the software under test, we stop processing new terms after we find the first error-revealing term.

We could make the process more efficient if we had a means of discarding non-error-revealing terms, *before* we have to evaluate them and the oracle. Suppose we define an equivalence relation $\sim$ on terms, and discard a new candidate term if it is equivalent to an old term. What properties should the equivalence relation have? The most important desired property of $\sim$ is that discarding a term based on the relation should not make us miss the error. We say that an equivalence relation is *safe* if it does not lead to discarding an error-revealing term. It is easy to see that a relation that satisfies the following *oracle consistency* property is safe:

$$t_1 \sim t_2 \Rightarrow O([\![t_1]\!]) = O([\![t_2]\!]). \tag{4.1}$$

Discarding terms based on an oracle-consistent equivalence relation $\sim$ is safe: if a term is equivalent to an old (thus non-error-revealing) term, oracle consistency implies that the new term is also not error-revealing. However, oracle consistency is not the only desired property of an effective equivalence relation: if it were, we could simply define term equivalence as term equality ($t_1 \sim t_2$ iff $t_1 = t_2$), which is easily shown to be oracle consistent but leads to few discards. The missing desirable property of an equivalence relation is *efficiency*: an efficient equivalence relation is both oracle-consistent and as *coarse* as possible, putting as many terms in the same equivalence class, and leading to the greatest number of discards. In fact, the *ideal* equivalence relation on terms would satisfy the following property:

$$t_1 \sim t_2 \text{ iff } O([\![t_1]\!]) = O([\![t_2]\!]). \tag{4.2}$$

Given such a relation, we could avoid evaluating and checking the oracle on *all* but the first term: if we are lucky and the first term reveals the error, we need not consider any other terms, and otherwise, to determine if a new candidate term reveals the error, we only need to check whether it is equivalent to the first term. Such a relation is ideal in the sense that it requires the fewest number of term evaluations and oracle checks (one), and it is guaranteed not to miss the error.

Equivalence for Pruning

The last section discussed the use of an equivalence relation for filtering out useless terms *after* they are generated using a term generation procedure. However, a procedure that generates many equivalent terms in the first place is wasteful: equivalent terms do not get us any closer to finding the error. It would be more efficient if we could use an equivalence relation to *prevent* the procedure from generating equivalent terms, rather than filtering already-generated terms. This section explores the use

```
Procedure DRT
Input: Operation set $ops$, time limit $t$, oracle $O$, legality predicate $Leg$,
       equivalence relation on terms $\sim$.
Result: An error-revealing term, or $\perp$ if none is generated within time limit.
1 $C \leftarrow \{\}$
2 while time limit $t$ not expired do
3     $img \leftarrow \{\ t = \langle op, t_1, \ldots, t_k \rangle \in I(C, ops)\ :\ Leg(op, t_1, \ldots, t_k)\ \wedge\ \underline{\forall t' \in C : t' \nsim t}\ \}$
4     $t =$ rchoose $img$;
5     if $\neg O([\![t]\!])$ then
6         return $t$
7     $C \leftarrow C \cup \{t\}$
8 return $\perp$
```

Figure 4-2: The final directed random testing algorithm DRT. It improves on BU/L by discarding equivalent terms from the image. This is reflected in the computation of the image subset *img*. The underlined text shows where equivalence pruning is performed.

of an equivalence relation by a bottom-up generator, to avoid generating equivalent terms.

Figure 4-2 shows procedure DRT ("directed random testing"), which extends BU/L by discarding terms based on an equivalence relation given as input. DRT takes five inputs: an operation set, a time limit, an oracle, a legality predicate, and an equivalence relation on terms. DRT combines term generation with evaluation and oracle checking, thus the additional input oracle. The procedure starts with an empty component set. It repeatedly and randomly selects and element from the component set's image, but first discards from the image terms that are illegal, as well as terms that are equivalent to a term already in the component set. If a term is error-revealing, DRT returns the term and halts the generation process. This behavior is appropriate under our current assumption that there is a single error in the software; it also makes the formal treatment below clearer. Section 4.2.3 addresses the presence of multiple errors.

What properties should an equivalence relation used by DRT have? Like in the previous section, we would like the relation to be *safe*—discarding a term based on the relation should not prevent DRT from discovering the error—and to be sufficiently *coarse*, allowing for the greatest possible number of discards. The rest of this section and Sections 4.2.2–4.2.4 discuss safety and coarseness in the context of DRT.

In the previous section, where we used the equivalence relation to discard already-generated terms, safety was simple to understand: an equivalence relation should not lead to an error-revealing term being discarded. The concept is trickier when using the equivalence relation to discard terms from the component set, because discarding a term from the component set can affect the scope of terms that may be generated in the future. For example, discarding the term `Rat(1,2)` from the component set may prevent the subsequent creation of `Poly(Mono(Rat(1,2),3))`; if the latter is the only error-revealing term, then `Rat(1,2)` should not be discarded. A definition of safety should express the idea that the "scope" of possible terms that can be created by DRT after discarding a particular term always contains an error-revealing term, if one exists. (Recall that we are currently assuming only one error exists, so it only matters that an error-revealing term exists, and do not care about the specific operation that reveals the error.)

Before formally defining safety, we first define the notion of an *execution* of DRT, which captures the state of the algorithm at some point in time, and then define the notion of *scope*, which describes the terms that DRT is capable of generating in a given execution. If at a point during the algorithm's execution a term goes out of scope, it means that there is no possible series of generation steps that can produce the term; in other words, the generator is no longer able to produce the term.

Definition (execution). An *execution* is a sequence of terms $\langle t_1, \ldots, t_n \rangle$ corresponding to the terms generated (line 4) during a partial or complete run of DRT, in the order that they were generated. If the execution represents a complete run, the last term in the sequence is an error-revealing term.

Note that because DRT takes as input an equivalence relation on terms, the definition of an execution is implicitly parameterized over an equivalence relation. As a simple example, using a trivial equivalence relation based on term equality ($t_1 \sim t_2$ iff $t_1 = t_2$), the sequence $\langle 1, \texttt{Rat(1,1)} \rangle$ is a valid execution of DRT. Assuming $O(\texttt{Rat(1,1)}) = \mathit{true}$, the sequence $\langle 1, \texttt{Rat(1,1)}, 2, \texttt{Rat(1,2)} \rangle$ is also an execution. On the other hand, the sequence $\langle 1, \texttt{Rat(1,1)}, \underline{\texttt{Rat(1,1)}} \rangle$ is not an execution: the algorithm would not select the last term, because it is equivalent to the second term. We say execution e_2 *extends* execution e_1 if e_1 is a prefix of e_2; we denote all the extensions of e as $\mathit{exts}(e)$.

We use the concept of an execution to define, at a given point in the algorithm's execution, the set of all terms that it is possible to generate in the future.

Definition (scope). The *scope* of an execution e is the set of terms that can be generated across all possible extensions of e, i.e. $\mathit{scope}(e) = \{t \mid t \in e', \text{ for some } e' \in \mathit{exts}(e)\}$.

Safety states that an error-revealing term is always in scope (if there exists an error-revealing term), regardless of the particular execution. Recall we are currently assuming that there is no more than a single error in the software under test, and we care only to find one term that is error-revealing. (Section 4.2.3 generalizes safety beyond this setting.) Also assume an operation set *ops*, unlimited time limit t, oracle O, ideal legality predicate *Leg*, and an equivalence relation $\sim$ on the terms in $\mathit{terms}(\mathit{ops})$.

Definition (safe equivalence on terms). The relation $\sim$ is *safe* iff for any execution e of DRT$(\mathit{ops}, t, O, \mathit{Leg}, \sim)$, if there is some term $t \in \mathit{terms}(\mathit{ops})$ such that $O(\llbracket t \rrbracket) = \mathit{false}$, then there is a term $t' \in \mathit{scope}(e)$ such that $O(\llbracket t' \rrbracket) = \mathit{false}$.

$$ops = \{\ \texttt{a:}\to\texttt{A},\ \texttt{b:A}\to\texttt{A},\ \texttt{c:A}\to\texttt{C},\ \texttt{d:}\to\texttt{D}\ \}$$

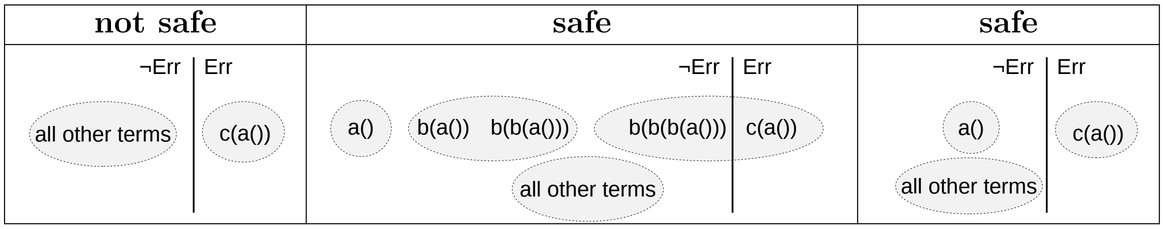

Figure 4-3: Example equivalence relations showing that oracle consistency is neither necessary nor sufficient to establish safety of Drt. The left relation is oracle consistent but not safe. The right relation is not oracle consistent but safe.

In other words, an equivalence relation is safe if, for any execution, there is always an error-revealing term in scope—assuming there is at least one error-revealing term in the universe. An equivalence relation on terms used by Drt should be safe: if it is not, it can render Drt unable to generate an error term on a given execution. (In practice, an unsafe relation can nevertheless be useful in guiding random generation; we discuss this in in Section 4.4.)

To illustrate the definition, Figure 4-3 shows three equivalence relations for an operation set where the term `c(a())` is the only error-revealing term. The first equivalence relation (left) is the "ideal" relation from last section: it contains two equivalence classes, one containing the single error-revealing term, and another containing all other terms. This relation is not safe. For example, any execution of Drt that starts by generating and adding to the component set the term `d()` will never be able to generate the error-revealing term `c(a())`, because `a()` (which is equivalent to `d()`) will never be in the component set. The example shows that oracle consistency, which was sufficient to establish safety when *filtering* terms, is not sufficient to establish safety when the relation is used for pruning terms.

The second equivalence relation (middle) shows, perhaps surprisingly, that oracle consistency is in fact not even *necessary* to establish safety. The relation puts an error-revealing and a non-error-revealing term in the same equivalence class, yet guarantees that the error term `c(a())` can always be generated, no matter what terms are discarded. The relation is safe because `a()` is in its own equivalence class, thus can always be added to the component set, and the term `b(b(b(a())))`, which is non-error-revealing but in the same equivalence class as the error-revealing term `c(a())`, can never be generated, and thus cannot mask the error-revealing term. The last example (right) shows a relation that is both oracle consistent and safe.

4.2.2 Establishing Safety

Safety captures the requirement that an equivalence relation should not cause Drt to miss an error. Proving an equivalence relation safe involves reasoning about all executions of Drt and the connection between the relation and the oracle. In this section, we describe two lemmas that can be used to break down the task of establish-

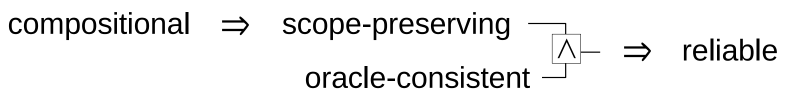

Figure 4-4: Lemmas used to establish that an equivalence relation on terms is safe.

ing safety into separate concerns. Figure 4-4 summarizes the properties and lemmas involved. As the figure shows, oracle consistency is one of the properties used to establish safety. We saw in the last section that oracle consistency is neither necessary nor sufficient to establish safety. However, together with *scope preservation*, it is sufficient. Scope preservation states that for *any* term (not just an error-revealing term), an equivalent term is always in the scope of Drt's execution. Again, assume an operation set *ops*, unlimited time limit t, oracle O, ideal legality predicate *Leg*, and an equivalence relation $\sim$ on the terms in $terms(ops)$.

Definition (scope-preserving). The relation $\sim$ is *scope-preserving* iff for any execution e and any term $t \in terms(ops)$, there is a term $t' \in scope(e)$ such that $t \sim t'$.

Figure 4-3 shows examples of relations that are and are not scope-preserving. The first relation (left) is *not* scope-preserving: the term `c(a())` is not is scope for every execution: in particular, if the first term generated in an execution is `d()`, then the term `c(a())` goes out of scope because `a()` will always be discarded (it is equivalent to `d()`), which makes it impossible to subsequently generate `c(a())`. It is easy to see that the two other relations (middle and right) are scope-preserving.

Lemma 4.1. If $\sim$ is scope-preserving and oracle-consistent, then $\sim$ is safe.

Proof. The fact that $\sim$ is safe follows easily. Suppose there is some error-revealing term in $terms(ops)$, and consider any execution e. We must show that there is an error-revealing term in $scope(e)$. Pick any term $t \in terms(ops)$ that is error-revealing. Since $\sim$ is scope-preserving, there is some $t' \in scope(e)$ where $t' \sim t$. Since $\sim$ is oracle-consistent and t is error-revealing, t' is also error-revealing. Thus, there is a term in $scope(e)$ that is error-revealing. ■

Lemma 4.1 gives us a way to break down the proof that an equivalence relation is safe into smaller proofs that deal with separate concerns: oracle consistency deals with the connection of the equivalence relation to the error terms, and scope preservation establishes that the equivalence relation does not "lose" any equivalence classes. Lemma 4.1's implication does not go the other way (as demonstrated by the middle example from Figure 4-3).

A second property, *compositionality of equivalence*, lets us further decompose a proof of safety by establishing a connection between $\sim$ and each operation. Compositionality states that using equivalent terms as inputs to an operation results in an equivalent term.

Definition (compositionality). Consider an operation set *ops* and equivalence relation $\sim$ on *terms*(*ops*). We say that $\sim$ is *compositional* iff for every operation $o : r_1 \times \ldots \times r_k \to r_{out}$ in *ops* and $1 \leq i \leq k$,

$$t_i \sim t'_i \to o(t_1, \ldots, t_i, \ldots, t_k) \sim o(t_1, \ldots, t'_i, \ldots, t_k).$$

Compositionality is attractive for two reasons. First, it is independent of the DRT algorithm. (Scope and safety are both defined in terms of an execution of DRT.) Second, it lets us reason locally about a single operation and a single composition step, and as the next lemma shows, this local reasoning can be lifted to reasoning about all the possible executions of DRT to establish scope preservation.

Lemma 4.2. A compositional equivalence relation is scope-preserving.

Proof. Suppose $\sim$ is compositional. We prove that for any term t and execution e, we can extend e to an execution e' that contains a term t' equivalent to t. Then, since *scope*(e) includes all the terms in e', it follows that $t' \in$ *scope*(e), establishing that $\sim$ is scope-preserving.

Now, we prove that for any term t and execution e, we can extend e to an execution e' that contains a term equivalent to t. The proof is by induction on the number of operations in t. For the base case, t has a single operation (i.e. it is a constant term). If e has already yielded a term equivalent to t, let $e' = e$. If not, let $e' = e \bullet \langle t \rangle$, which represents a valid execution of DRT (the "$\bullet$" operator denotes sequence concatenation). In both cases, e' contains a term equivalent to t. (The base case does not require compositionality.)

For the induction step, suppose that for all terms of size less than N, we can extend an execution to one containing an equivalent term. We show that the same is true for any term of size N. Let $t = op(t_1, \ldots, t_k)$ be such a term, and e be an arbitrary execution. By the induction hypothesis, we can extend e to an execution e_1 containing a term $t'_1 \sim t_1$, then extend e_1 to an execution e_2 containing a term $t'_2 \sim t_2$, etc. until we reach an execution e_k that contains terms $t'_1, \ldots, t'_k$, which are equivalent to $t_1, \ldots t'_k$ respectively. Let $t' = op(t'_1, \ldots, t'_k)$ and $e' = e_k \bullet \langle t' \rangle$. By compositionality (applied k times, once per input terms, since compositionality only applies to one input term), $t' \sim t$, so e' is an execution that contains a term equivalent to t. ■

In summary, we can establish that an equivalence relation $\sim$ is safe by establishing that it is compositional and oracle-consistent, properties that allow us to reason independently about the equivalence relation's connection to each operation and to the oracle.

4.2.3 Multiple Errors

We have developed DRT assuming we care only to obtain one term that causes the oracle to fail. This suffices if there is at most one error in the software under test.

$$ops = \{\ \texttt{a}{:}\rightarrow\texttt{A},\ \texttt{b}{:}\texttt{A}\rightarrow\texttt{A},\ \texttt{c}{:}\texttt{A}\rightarrow\texttt{C},\ \texttt{d}{:}\rightarrow\texttt{D}\ \}$$

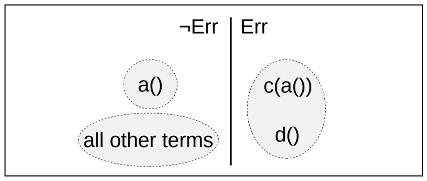

Figure 4-5: An equivalence relation that is safe according to the definition of safety from Section 4.2.1, but may prevent the generation of the error-revealing term `d()`.

Suppose the opposite is true: we care to obtain as many different error-revealing terms as possible—perhaps because each distinct term may reveal a different error.

A simple modification to DRT would enable it to return more than one term. When it discovers an error-revealing term, instead of returning it and halting, DRT adds the term to a set of terms *Errors* and continues the generation process. When the time limit expires, DRT returns the set *Errors*.

The above modification means that any error-revealing term that is generated will be output. However, another obstacle stands in the way of generating multiple error-revealing terms: our current definition of safety stipulates only that *some* error-revealing term be in scope, which is may not be sufficient to ensure that distinct errors are revealed For example, consider the relation shown in Figure 4-5. There are two error-revealing terms, `c(a())` and `d()`, both in the same equivalence class. The relation is safe according to our current definition of safety: the term `c(a())` can be generated in any execution. However, only one of the two error-revealing terms will be generated in an execution, because the two error-revealing terms are equivalent.

Fortunately, a small modification to the definition allows us to define safety for the case where we are interested in generating more than one term. First, we generalize the notion of an oracle to distinguish among different error-revealing values. Let *ErrorClasses* be an arbitrary set of symbols; we use elements of the set to classify error values. A *distinguishing oracle* is a *function* (not a predicate) $O : \mathit{Vals} \rightarrow \mathit{ErrorClasses}$ that maps a value to an *error class* $c \in \mathit{ErrorClasses}$. A distinguishing oracle generalizes the concept of an oracle: we can convert a "traditional" oracle into a distinguishing oracle by letting $\mathit{ErrorClasses} = \{\mathit{true}, \mathit{false}\}$. However, using a distinguishing oracle lets us group error values into error classes, to signal that error values from different classes are considered different, and should not be masked by the equivalence relation. For example, an oracle that maps every error value to a different class means that every error value is considered to be a different error. As a convention, we always use the error class *true* to represent non-error values.

A definition of safety using distinguishing oracles is very similar to our original definition. Assume an operation set *ops*, unlimited time limit t, a distinguishing oracle O, ideal legality predicate *Leg*, and an equivalence relation $\sim$ on the terms in $\mathit{terms}(\mathit{ops})$. The modified parts of the definition are underlined.

Definition (safe equivalence on terms with distinguishing oracle). The relation $\sim$ is *safe* iff for any execution e of $\textsc{Drt}(ops, t, O, Leg, \sim)$, if there is some term $t \in terms(ops)$ such that $\underline{O(\llbracket t \rrbracket) = c}$ (where $c \neq true$), then there is a term $t' \in scope(e)$ such that $\underline{O(\llbracket t' \rrbracket) = c}$.

Other definitions from Section 4.2.1 remain unchanged.

Referring back to the example from Figure 4-5, if we are interested in distinguishing between the two error-revealing terms, we stipulate that the equivalence relation be safe with respect to an oracle that puts the two error-revealing terms in different error classes. The relation shown in the figure is not safe under such an oracle.

4.2.4 Efficiency

Earlier in the chapter we argued that safety alone does not yield a useful equivalence notion; the relation should also be coarse, because larger the equivalence classes result in greater potential for discarding terms. It takes little effort to define an equivalence relation that is safe but not coarse (e.g. term equality) or one that is coarse but not safe (e.g. the relation where every term is equivalent). Defining an equivalence relation, whether by hand or via program analysis, typically involves a tradeoff between safety, coarseness, and human effort. For example, we could spend time studying the software under test, and define an equivalence relation that we can prove is safe and also coarse; this would likely require significant effort.

An automated technique that derives an equivalence relation is likely to sacrifice efficiency, safety or both. Section 4.4 describes heuristic pruning, a technique that automates equivalence and can suffer from a lack of both safety and efficiency, but is nevertheless effective in practice.

4.2.5 Discarding Terms and Knowledge of Errors

Throughout this section, we have stated as a matter of fact that discarding equivalent terms is always beneficial, i.e. it increases the chances that DRT will generate an error-revealing term. Strictly speaking, this may not be true. For example, suppose that we knew that the error-revealing terms are exactly those terms that evaluate to the polynomial "$2x$." In that case, keeping in the component set more than one equivalent `Rat` term that represents the rational coefficient "2" (e.g. `Rat(2,1)`, `Rat(4,2)`, `Rat(6,3)`, etc.) may actually *increase* the odds of discovering the error (the chances of creating a polynomial with exponent 2 would increase).

Another way of viewing this problem is by observing that the effect of discarding equivalent terms makes DRT's generation uniform over the space of equivalent terms (recall that the **rchoose** operator selects uniformly at random). Depending on the location of the errors, a different distribution could result in increased probability of revealing an error. However, lacking advance knowledge about the location of errors in the space of terms or values (as is usually the case), uniform selection among equivalent terms is a reasonable choice.

method	Leg_{method}
`Poly(Mono... ms)`	$\texttt{ms} \neq \texttt{null} \wedge$ $(\forall i : 0 \leq i < \texttt{ms.length} : \texttt{ms}[i] \neq \texttt{null}) \wedge$ $(\forall i : 0 \leq i < \texttt{ms.length} : \texttt{ms}[i]\texttt{.coeff.num} \neq 0) \wedge$ $(\forall i : 1 \leq i < \texttt{ms.length} : \texttt{ms}[i-1]\texttt{.exp} <= \texttt{ms}[i-1]\texttt{.exp})$
`Poly.add(Mono m)`	$\texttt{m} \neq \texttt{null}$
`Mono(Rat c, int e)`	$\texttt{c} \neq \texttt{null} \wedge \texttt{e} \geq 0$
`Rat(int n, int d)`	$\texttt{d} \neq 0$
`Rat.approx()`	*true*
`add(Rat r)`	$\texttt{r} \neq \texttt{null}$

Figure 4-6: Legality predicates for polynomial data types.

class	*For* o_1, o_2 *of the given class* $o_1 \sim o_2$ *iff:*
`Rat`	$o_1\texttt{.num} = o_2\texttt{.num} \;\wedge\; o_1\texttt{.den} = o_2\texttt{.den}$
`Mono`	$o_1\texttt{.coeff} \sim o_2\texttt{.coeff} \;\wedge\; o_1\texttt{.exp} = o_2\texttt{.exp}$
`Poly`	$o_1\texttt{.ms.length} = o_2\texttt{.ms.length} \wedge \forall i : 0 \leq i \leq\; o_1\texttt{.ms.length} :$ $o_1\texttt{.ms}[i] \sim o_2\texttt{.ms}[i]$

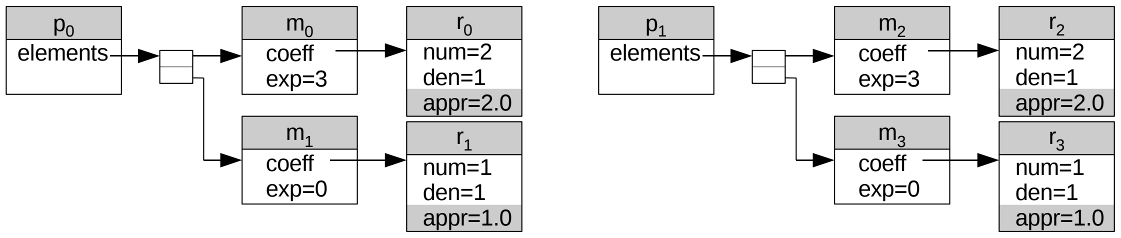

Figure 4-7: Top: an equivalence relation on polynomial *values*. Bottom: two values that are equivalent according to the definition.

4.3 Example

This section grounds the concepts we have presented in the chapter by applying directed random testing to the polynomial operations in `Rat`, `Mono` and `Poly` and oracle from Figure 2-2, reproduced for convenience in Figure 4-8. First we define a legality predicate and an equivalence relation. Figure 4-6 shows the legality predicate, separated for each operation, translated from the method comments in Figure 1-1. Establishing the predicate to be ideal, i.e. that for every operation *op* it returns *true* on $Leg(op, t_1, \ldots, t_k)$ if and only if $[\![t_i]\!] \in LV_i^{op}$ for $1 \leq i \leq k$, is made difficult by the fact that the only documentation of preconditions is found in the informal comments. (This is not an uncommon problem in practice.) In our case, the methods and preconditions are small enough that a simple inspection can convince us that they are accurately captured by the predicates in Figure 4-6.

The next step is to define an equivalence relation. We do this by defining an equivalence relation on *values*, and lifting the relation to terms by saying that two

```
public class Poly {
 boolean repOk() {
   for (int i=1 ; i<ms.length ; i++) {
     if (ms[i-1].expt <= ms[i].expt)
       return false;
     if (ms[i].coeff.numer == 0)
       return false;
   }
   return true;
 }
}
```

Figure 4-8: Code representing the polynomial value oracle (reproduced for convenience from Figure 2-2).

terms are equivalent iff the values they evaluate to are equivalent. Figure 4-7 shows the equivalence relation on values of the three polynomial data types. The relation is defined recursively. For example, equivalence of `Poly` values is defined in terms of equivalence of `Mono` values.

We can establish that the relation is safe by establishing that it is oracle-consistent, and that it is compositional for each method. To illustrate at least one part of the proof process, we informally argue that it is oracle-consistent for the oracle `InvariantChecker.check(Poly)` (Figure 4-8).

Lemma. The equivalence relation $\sim$ given in Figure 4-7 is oracle-consistent with respect to the oracle `InvariantChecker.check(Poly)`.

Proof (informal). Suppose t_1 and t_2 are equivalent, $[\![t_1]\!] = p_1$, and $[\![t_2]\!] = p_2$. Then, by the definition of equivalence, the `ms` fields of p_1 and p_2 have the same length. If their length is 0, the oracle returns `true` on both (the for-loop is never entered). Suppose their length is greater than 0, and the oracle returns `false` for one of the polynomials. This means that for some i, either the first or second if-clause in the for-loop of the oracle evaluates to false. Then, by the definition of equivalence, it follows easily that for the other polynomial, the for-loop also evaluates to false for the same index i, and the oracle returns `true` for the other polynomial as well. The argument for the oracle returning `false` is analogous. ■

A proof of compositionality for each method would follow a similar structure; assuming that the inputs to a given method are are equivalent, we must show that the result will also be equivalent. For example, we could prove that compositionality holds for the receiver argument of method `Poly.add(Mono)` (recall that compositionality is over an individuals argument) by assuming two equivalent `Poly` terms t_1 and t_2, and showing that t_1`.add(`m`)` $\sim$ t_2`.add(`m`)`.

Using these definitions of legality and equivalence, a directed random test generator avoids generating many useless terms. For example, no terms with the illegal sub-term `Rat(1,0)` are ever generated, because the term is never added to the component set. The generator also avoids generating equivalent terms like `Rat(1,1)` and

Legal, distinct `Poly` objects generated

length	1	2	3	4	5	6	7	8	9	10	> 10
DRT	126	133	68	26	8	3	4	0	10	1	0
TD	141	70	3	0	0	0	0	0	0	0	0

Figure 4-9: Number of legal, non-equivalent polynomials of varying lengths generated by DRT and TD in one minute.

`Rat(2,2)`, or equivalent compositions like `Mono(Rat(1,1),1)` and `Mono(Rat(2,2),1)`, because the component set would never contain both `Rat(1,1)` and `Rat(2,2)`. On the other hand, the generation process is constrained to terms already in the component set, and as we discussed in Chapter 3, this could in result in repetitive test inputs.

To understand DRT's performance versus another random testing technique, we performed an experiment using implementations of DRT and TD to generate `Poly` objects for 1 minute. We used a larger number of seeds terms, including the integers -100, -10, -1, 0, 1, 10, and 100. We also used operations not shown in the example, including methods that add, subtract, and multiply polynomials.

In one minute, DRT generated approximately 10,000 distinct, legal terms, of which 380 were of type `Poly`. In the same amount of time, TD also generated approximately 10,000 terms, all of type `Poly`, but only 200 of them legal and distinct. (These results are averages over 10 runs, executed in a 2.4 GHz Intel Core 2 Duo with 2GB memory.) Despite the fact that DRT is less focused on generating terms for the sort of interest, the procedure is able to generate more useful `Poly` terms.

DRT also generated a larger variety of polynomial lengths (number of monomials) than TD. Figure 4-9 shows the distribution of polynomial lengths for a 1-minute run of DRT and TD. The probability that TD generates a legal polynomial of some large length N remains constant throughout the execution of the procedure; on the other hand, the probability that DRT generates a legal polynomial of length N increases as the component becomes populated with more legal, distinct monomials and polynomials.

4.4 Heuristic Pruning

The first key idea in directed random testing is to combine random generation with pruning of illegal and redundant inputs. The second key idea is pruning based on automatically-derived (if potentially inaccurate) predicates. The aim of heuristic pruning is to increase directed random testing's cost effectiveness by making it fully automatic.

Heuristic pruning determines term legality and equivalence based on the runtime behavior of the software under test. To determine term legality, the generator executes the term. If the term executes normally, the generator heuristically determines that the term is legal. The definition of "normal execution" may depend on the particular programming context. To determine the equivalence of two runtime values,

```
 public class Mono {

  public Mono(Rat coeff, int exp) {
    if (coeff == null || exp < 0)
      throw new IllegalArgumentException("Invalid arguments.");
    this.coeff = coeff;
    this.exp = exp;
  }

  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null  || !(o instanceof Mono)) return false;
    Mono m = (Mono)o;
    return this.coeff.equals(m.coeff) && this.exp == m.exp;
  }
}
```

Figure 4-10: Excerpts from class `Mono` showing code constructs used in heuristic pruning.

the generator observes their state directly or indirectly, e.g. via observers defined in the software under test.

RANDOOP, a directed random testing tool for Java described in Chapter 7, judges an execution normal if it raises no exceptions. To determine object equivalence, RANDOOP leverages the notions of object equality already present in the software: it calls two values equivalent if they have the same type and calling that type's `equals` method on the two objects returns `true`. An example of a class for which these heuristics work well is class `Mono`. Figure 4-10 shows how `Mono`'s constructor checks the validity of its arguments and throws an `IllegalArgumentException` if the constructor's preconditions are violated. The `equals` method in `Mono` is consistent with the notion of equivalence from the previous section: it returns `true` when the two monomials' coefficients are equivalent and their exponents are the same.

Depending on the strategy to determine legality or equivalence, and on the software artifact under test, heuristic pruning may be inaccurate. Procedures can fail to check the legality of their inputs due to programmer omission or performance considerations, causing operations to be passed invalid inputs and not exhibit externally-observable abnormal behavior. Equivalence checking based on observer procedures like `equals` can yield inaccurate results if the procedure ignores portions of a value's state relevant for an operation, or if observer procedures contain errors. The key hypothesis behind heuristic pruning is that despite its probable inaccuracy, it can usefully prune the input space and increase the ability of directed random testing to achieve greater coverage and reveal errors. We validate this claim experimentally in Chapter 8.

Chapter 5

Directed Random Testing for Sequences

Terms are a natural language for describing the creation of values using operations. Up to now, we have viewed operations as functions that map input values to an output value. In a purely functional programming language (e.g. Haskell [20]), this functional model captures all possible behaviors obtainable via operations, and terms are a perfectly matched language for test input generation. However, for non-functional programming languages like C++, Java, or C#, a simple term language is limited in expressiveness and cannot express important behaviors, including patterns of mutation and aliasing that can arise by chaining together sequences of operations. For software written in these languages (which is the majority of software), generating inputs that are terms (i.e. operation trees, as opposed to sequences) restricts testing to a subset of all possible behaviors. This limitation can prevent a test generator from revealing important errors.

This chapter extends the ideas behind directed random testing to a more expressive language of *operation sequences*. We begin by describing expressiveness deficiencies of terms. In a nutshell, terms are unable to express certain aliasing patterns, such as using the same object reference twice as input to method that takes more than one input. Terms are also unable to express patterns of side effects (i.e. the modification by an operation to the state of its inputs), for example particular scenarios where a sequence of operations is performed using a value as input multiple times not for the values the operations return, but for the side effects of the operations on the value.

After discussing these problems with terms, We show that *sequences* of operations are able to express patterns of behavior inexpressible with terms. We revisit the ideas that make up directed random testing, including bottom-up generation, legality, equivalence, and heuristic pruning, adapting the ideas to work on operation sequences instead of terms. Finally, we discuss two problems that arise when trying to adapt equivalence from terms to sequences, and present *index pruning*, which addresses one of these problems.

Description of scenario	Test input that creates scenario	Reason(s) scenario cannot be created with a term.
A call of `Mono.equals` is made on the same object.	`Mono m = new Mono();` `m.equals(m);`	Cannot refer to `m` twice in the same operation.

Figure 5-1: An important aliasing scenario impossible to create with a term.

5.1 Deficiencies of Terms

Terms are insufficient to express patterns of aliasing and side effects that can arise using a set of operations. These problems arise because in most languages, operations do not match the traditional functional interpretation of terms. For instance, in Java, the same method or constructor invocation (e.g. "`new Rat(1,1)`") returns two distinct values (objects) when invoked twice on *identical* inputs. In other words, `Rat` is *non-deterministic.* Also, a Java method may mutate the state of its inputs, a possibility not accounted for by the functional interpretation of terms. This section discusses concrete examples that illustrate patterns of aliasing and mutation inexpressible with terms, and why this can decrease the error-revealing effectiveness of a test generation procedure based on terms.

Non-Determinism and Aliasing

Aliasing refers to the situation where two symbolic names refer to the same region in memory. Aliasing can arise in most programming languages, and programmers often write code to account for the possibility of aliasing. For example, recall the `equals` method of the `Mono` class from Figure 4-10:

```
public boolean equals(Object o) {
  if (this == o) return true;
  if (o == null  || !(o instanceof Mono)) return false;
  Mono m = (Mono)o;
  return this.coeff.equals(m.coeff) && this.exp == m.exp;
}
```

The if-condition in the first line in the body of `equals` ("`if (this == O) ...`") is an aliasing check: it checks if the memory location that the receiver object `this` refers to is the same location that the argument `o` refers to. If it is, `this` and `o` are equal (in fact, identical), so `equals` returns true. Figure 5-1 shows a test input that drives execution through the true branch of the if-condition. While in this example the code following the true branch is simple and obviously correct, it could easily contain a larger or more complex body of code, making it important to include a test input that follows the branch.

Unfortunately, no term over polynomial operations would drive execution through this branch, because to do so it would have to specify the *same object* (which is different from the *same term*) twice as input to method `equals`. In a term, if we wish to specify the same value twice as input, we do this by replicating the same sub-term, e.g. `equals(Mono(),Mono())`, because under the functional interpretation of terms (Section 3.3), equal terms result in identical values. The tactic does not work for `Mono()` (or for any other polynomial operation that returns an object) because the operations are not functions: two calls of `Mono()` in a test input result in two distinct objects.

Notice that the test input in Figure 5-1 that manages to follow the true-branch of the aliasing check does this by assigning a name `m` to the result of the constructor call, and using the name to refer to the value in the next method call. The language of operation sequences we develop in Section 5.2 gives a name to the value resulting from an operation, and uses the name to specify that the value be used as input to later operations, allowing aliasing patterns to be expressed.

Side Effects

A side effect refers to the situation where an operation modifies the state of one or more of its inputs (or any reachable program state, including global program state). Terms cannot express some patterns of mutation that may be important to adequately test a software artifact. To illustrate this problem, we augment our running polynomial example with a new data type and new operations. Figure 5-2 shows class `PStack`, which implements a stack of polynomials. A `PStack` object holds a collection of `Poly` objects that can be accessed using stack operations `push(Poly)` and `pop()`. In addition, a polynomial stack provides operations to perform polynomial arithmetic on the top elements of the stack. For example, the `add()` method pops two polynomials from the stack, adds them, and pushes the result on the stack. Like the polynomial data types, `PStack` is based on a real programming assignment given to undergraduate computer science students at MIT.

Unlike the data types we have seen so far, `PStack` has *mutable* state: the value of its fields can change as a result of an operation. Thus some stack methods, in addition to returning a value, have an additional *side effect*: they modify the state of the stack. For example, the `pop()` operation, in addition to returning a polynomial object, modifies the stack's state, decrementing the value of the `size` field by 1.

Side effects present a representational problem for terms. In the absence of side effects, it is natural to think of the result of an operation as its return value. With side effects, the result of the operation is less clear: is the result if `pop()` a polynomial, or a "new stack" in a different state from the original? Our term language from previous chapters always designates the result of an operation as the return value. In the case of the polynomial stack class, this designation makes it impossible for an input generation to generate important values. Figure 5-3 shows examples of mutation-related scenarios that cannot be created using terms. The first example demonstrates that a very basic scenario—pushing more than one element on the stack—is impossible to express with terms, because the return value of a `push` operation is a `boolean` value.

Other researchers have attempted to overcome this problem by adding syntax that make it possible for the result of term to be interpreted as either the result of the operation, or one of its (possibly-mutated) input arguments. For example, Henkel

```java
public class PStack {

  // representation invariant:
  //   stack != null,
  //   0 <= size <= stack.length
  private Poly[] stack;
  private int size;

  // Construct a new stack.
  // Initial capacity will be 10.
  public PStack() {
    this(10);
  }
  // Construct a new stack with
  // the given capacity.
  public PStack(int capacity) {
    stack = new Poly[capacity];
    size = 0;
  }

  // Pushes p on the stack.
  // Returns true if the value
  // was added successfully.
  public boolean push(Poly p) {
    if (size == stack.length)
      return false;
    stack[size-1] = p;
    size++;
    return true;
  }
```

```java
  // Pops top of the stack.
  // Requires: stack is not empty.
  public Poly pop() {
    return stack[--size];
  }

  // Pops the two top elements of
  // the stack, adds them, and
  // pushes result on the stack.
  // Requires: size of stack >= 2.
  public void add() {
    // ...
  }

  // Pops the two top elements of
  // the stack, subtracts them, and
  // pushes result on the stack.
  // Requires: size of stack >= 2.
  public void sub() {
    // ...
  }
  public PStack copy() {
    PStack s2 = new PStack();
    s2.stack = stack;        // ERROR
    return s2;
  }
}
```

Figure 5-2: A polynomial stack class `PStack`, used in the implementation of a stack-based polynomial calculator.

and Diwan [50] propose a term-based representation for Java that augments the term language with new notation: given a term $t = m(t_1, \ldots, t_k)$, $state(t)$ refers to the state of the first argument (i.e. the receiver) instead of the return value. With this new notation, it becomes possible to write a term that describes consecutive pushes on a stack, as demonstrated by the term

$$\texttt{push(}\mathit{state}\texttt{(push(PStack(), Poly())), Poly())}.$$

Unfortunately, the notation only partially solves the problem. Figure 5-3 (second example) shows an important error-revealing scenario that is not expressible using Henkel and Diwan's augmented notation. The scenario creates a new stack `s1` and pushes a polynomial on it. It then copies the stack, obtaining a new stack `s2`, pops `s2`'s top element, and then attempts to pop `s1`'s top element, resulting in an `IndexOutOfBounds` exception on the last pop. The exceptional execution cannot be induced by a traditional term, nor by Henkel's augmented terms, because it is impossible to refer to stack `s1` more than once after the `copy` operation. Henkel's

Description of scenario	Test input that creates scenario	Reason(s) scenario cannot be created with a term.

Two polynomials are pushed on the stack.	`Poly p = new Poly();`
	`Rat r = new Rat(1,1);`
	`Mono m = new Mono(r,1);`
	`Poly p2 = new Poly(m);`
	`PStack s = new PStack();`
	`s.push(p);           // *`
	`s.push(p2);`
		Cannot refer to `s` after `*`.

A stack's state is (erroneously) modified by an operation on a copied stack.	`PStack s1 = new PStack();`
	`PStack s2 = s1.copy();// *`
	`Poly p1 = new Poly();`
	`boolean b1 = s1.push(p);`
	`Poly p2 = s2.pop();`
	`Poly p3 = s1.pop();`
		Cannot refer to `s1` after `*`.

Figure 5-3: Two Important mutation scenarios impossible to create with terms.

augmented terms suffer from the same problem as traditional terms: it allows only a single use of a value.

The representational problems we have shown are all due to the inability to refer to the result of an operation more than once in later operations. The following section describes a language of operation sequences that assigns names to the result of each operation, allowing for multiple reference.

5.2 Operation Sequences

In order to express scenarios not expressible using terms, we define a language of operation sequences that lets us to refer to the result of an operation multiple times. An operation sequence represents a series of operations applications, where the inputs to each operation can be values returned by previous operations. All the test inputs in Figures 5-1 and 5-3 are actually operation sequences, demonstrating that operation sequences can represent scenarios impossible to represent with terms.

To be able to refer to the result of an operation multiple times, we develop a naming mechanism that identifies the result value of an operation in a sequence. We assume an infinite universe $Names$ of symbols. We associate a unique name with each operation in a sequence, which we use to refer to the result of the operation. Using this mechanism, we define a sequence of operation invocations as a sequence of triples, with each triple $(op, inputs, output)$ consisting of an operation $op \in ops$, a list of symbols $inputs \in Names^k$ specifying the inputs to the operation, and a new symbol $output \in Names$ specifying the result of the operation. We denote the i-th element of a sequence s as $s[i]$, and its operation, inputs and output names as

$s[i].op$, $s[i].inputs$ and $s[i].output$ respectively. The *names* of a sequence s, denoted $names(s)$, consist of the set of all names that are outputs for some operation in the sequence; because each name is unique, there are as many names as there are operations in the sequence. The sort of name n in sequence s, denoted $sort(n,s)$, is the result sort of the operation whose result the name represents. The index of name n in the sequence, denoted $idx(n,s)$, is the index of the operation that the name is associated with.

More formally, we define the set of *operation sequences* over an operation set *ops* inductively, as the smallest set satisfying the following properties.

1. The empty sequence is an operation sequence.
2. Consider:
   (a) An operation sequence s,
   (b) an operation $op = opname : r_1 \times \ldots \times r_k \to r_{out}$,
   (c) Names $n_1, \ldots, n_k \in names(s)$, where $sort(n_i, s) = r_i$ for $1 \leq i \leq k$, and
   (d) A name $n_{out} \notin names(s)$.

   Then $s \bullet \langle op,\ \langle n_1, \ldots, n_k\rangle,\ n_{out}\rangle$ is an operation sequence over *ops*.

For simplicity, we may write a sequence as abbreviated pseudocode, using primitives directly as operation arguments, e.g.

```
n3 = Rat(2,3) ; n5 = Mono(n3,1)
```

instead of the proper but more verbose sequence of triples

$$\langle\langle 2, \langle\rangle, \texttt{n1}\rangle,\ \langle\langle 3, \langle\rangle, \texttt{n2}\rangle,\ \langle \texttt{Rat}, \langle \texttt{n1}, \texttt{n2}\rangle, \texttt{n3}\rangle,\ \langle 1, \langle\rangle, \texttt{n4}\rangle,\ \langle \texttt{Mono}, \langle \texttt{n3}, \texttt{n4}\rangle, \texttt{n5}\rangle.$$

Shortly, we show that specific names in a sequence do not affect its interpretation, e.g. the sequence "`n3 = Rat(2,3) ; n5 = Mono(n3,1)`" is equivalent to

```
r = Rat(2,3) ; m = Mono(r,1).
```

5.3 Interpreting Sequences

The interpretation of operations in the world of terms (Section 3.3) associates with an operation $op = opname : r_1 \times \cdots \times r_k \to r_{out}$ a function

$$M_{op} : V_{r_1} \times \ldots \times V_{r_k} \to V_{r_{out}}$$

that takes as input k values and returns one value. However, as we saw in previous sections, an operation can have side effects on other values, including values other than those used as input to the operation.

To motivate our new interpretation of operations, recall the operation sequence from Figure 5-3, which creates two stacks that share a single polynomial array:

```
1.     PStack s1 = new PStack();
2.     PStack s2 = s1.copy();
3.     Poly p1 = new Poly();
4.     boolean b1 = s1.push(p);
5.     Poly p2 = s2.pop();
6.     Poly p3 = s1.pop();
```

Consider the last operation, `s1.pop()`. It takes as input a stack and returns a polynomial. As we saw in Section 5.1, the last invocation of `pop` also has as a side effect: it modifies the two stack values represented by `s1` and `s2`.

Another way of viewing `pop` is as an operation that takes as input the *state* of a program consisting of 6 values (one for each value in the sequence), and returns a *new* (different) state, consisting of 7 values (including the new return value of `pop`). The values in the new state have the same sort as they did in the previous state, but the values themselves may be different. In other words, the `pop` operation takes as input a state $\langle s_1, s_2, p_1, b_1, p_2, \rangle$, and returns a new state $\langle s'_1, s'_2, p_1, b_1, p_2, p_3 \rangle$, that contains a new value (p_3) returned by the operation, and two modified stacks ($s'_1 \neq s_1$ and $s'_2 \neq s_2$). In summary, viewing an operation as a *function on states* lets us account for the possibility of side effects on other portions of the state. (Although not shown in the above example, viewing an operation as a function on states also lets us account for aliasing: a tuple may have multiple elements with different names in the sequences that have the same value.)

As a second example, consider the sequence `r1 = Rat(1,1) ; r2 = Rat(1,1)`. Viewing `Rat`'s constructor as an operation that takes as input a state and returns a new state, the first application takes as input the empty state $\langle\rangle$ and returns a new state $\langle r_1 \rangle$ with a single `Rat` value. The second operation takes as input the state $\langle r_1 \rangle$, and returns a new state $\langle r_1, r_2 \rangle$, where $r_1 \neq r_2$. Thus, viewing an operation as a function on states also lets us account for the fact that the same application can return different values, depending on the context of the application.

The above examples illustrate the benefit of a state-based interpretation of operations. Next we make the interpretation of an operation sequence more precise. Call a finite sequence of values a *state*, and let *States* be the set of all possible states. We associate with each operation $op = opname : r_1 \times \cdots \times r_k \rightarrow r_{out}$ a function

$$M_{op} : States \times \mathbb{N}^k \rightarrow States$$

that takes as input a state and k *input indices*, and outputs a new state. The purpose of the indices is to specify the values in the input state that are used as immediate arguments to the operation. For example, letting $st = \langle p_1, p_2, m \rangle$ be a state consisting of two `Polys` p_1 and p_2 and one `Mono` m, and letting op be the `Poly.add(Mono)` operation, the application $M_{add}(st, 1, 3)$ refers to the addition of p_1 and m, and the application $M_{add}(st, 2, 3)$ refers to the addition of p_2 and m. Naturally, we restrict an operation to produce states with compatible sorts: if $M_{op}(st, i_1, \ldots, i_k) = st'$, then elements at the same index in st and st' have the same sort.

Given the new interpretation of operations, we can define the interpretation of a sequence. We view a sequence of operations as representing the state obtained by the repeated application of functions given by the operations in the sequence. The *state for sequence* s, denoted $[\![s]\!]$, is defined as follows:

- If s is the empty sequence, then $[\![s]\!] = \langle\rangle$ (the empty state).
- If $s = s' \bullet \langle op,\ \langle n_1, \ldots, n_k\rangle,\ n_{out}\rangle$, then $[\![s]\!]$ is $M_{op}([\![s']\!], idx(n_1, s'), \ldots, idx(n_k, s'))$.

We also say that a sequence s *evaluates to* or *produces* $[\![s]\!]$. To refer to the i-th element in the state, we write $[\![s, i]\!]$.

Note that the particular names in a sequence do not affect its interpretation, only the relation of the names to the results of specific operations. For example, given a fixed interpretation of polynomial operations, the sequences `i=1; j=2 ; r=Rat(i,j)` and `a=2; j=2; x=Rat(a,j)` evaluate to the same state, even though they have different names.

5.3.1 Error-Revealing Sequences

In Chapter 4 we assumed that the user provides an oracle O that determines if a value is error-revealing. From here on, we assume that an oracle O takes as input a *sequence* of values and determines if the sequence is error-revealing. Such an oracle matches better with sequences. It is also more powerful: an oracle that operates on a sequence of values can check correctness properties over relationships between the values. A common example, for object-oriented languages, is the `equals/hashCode` property, stating that two objects that are equal should have the same hash code. (Note that an oracle that operates on single values can be converted to a value-sequence oracle that operations on each element individually.)

The rest of the chapter adapts directed random testing to operation sequences. Much of the adaptation consists simply of recasting definitions and algorithms in the new vocabulary of sequences. However, Section 5.7 shows a problem that can make pruning based on equivalent sequences less effective than pruning based on equivalent terms. Section 5.8 discusses index pruning, a new pruning strategy that addresses the problem.

5.4 Composing Sequences

We initially proposed terms as a generation language because they naturally express the *compositional* nature of generating values using a set of operations. For example, both TD and BU generate a term of a given sort r by composing sub-terms; the sub-terms are recursively generated, or found in a component set. For example, we can create a `Poly` term via the operation `Poly` : `Mono...` $\rightarrow$ `Poly`, by applying the operation to sub-terms `Mono(Rat(2,1),3)` and `Mono(Rat(1,1),1)` to obtain

```
Poly(Mono(Rat(2,1),3),Mono(Rat(1,1),1).
```

We can also think of sequences as compositional entities that can be combined to generate a value of a desired sort. For example, to generate a sequence that produces a value of sort r, we can find an operation $opname : r_1 \times \ldots \times r_k \rightarrow r_{out}$, then recursively generate (or, as we show shortly, find in a component set of *sequences*) one or more sub-sequences that, when concatenated, yield a sequence that produces values for all the desired input sorts $s_1, \ldots, s_k$, and finally, append a call of the operation to the concatenated sequence, selecting appropriate names as input arguments. For example, we can build a sequence that creates a `Poly` via the operation `Poly` : `Mono...` $\rightarrow$ `Poly`, by (1) concatenating the two sequences "`r1=Rat(2,1); m1=Mono(r1,3)`" and "`r2=Rat(1,1); m2= Mono(r1,1)`," and (2) appending a call of `Poly(Mono...)` to the end of the concatenated sequences, resulting in

```
r1=Rat(2,1); m1=Mono(r1,3); r2=Rat(1,1); m2=Mono(r2,1); Poly(m1,m2).
```

Thus, sequences can be composed just like terms; they also have the additional benefit of greater expressiveness.

The notion of *concatenating* sequences is straightforward. The only complication arises from possible name clashes: if two sequences have one or more common names, concatenating them may not result in a valid sequence. For example, concatenating the sequences `r = Rat(2,3)` and `r = Rat(2,1)` would result in the sequence `r = Rat(2,3); r = Rat(2,1)`, which is not a valid operation sequence, because it does not have unique names in every index. To solve this problem, we assume that the concatenation operation for sequences, before concatenating two sequences, "renames" the names in all the input sub-sequences so that every sub-sequence has a unique set of names. Recalling from Section 5.3 that the result of evaluating a sequence is not affected by the specific names used, it is easy to see that this version of concatenation yields the expected result.

5.5 Directed Random Testing for Sequences

Having defined a notion of sequence composition, we can adapt directed generation to the domain of sequences. Figure 5-4 shows the modified directed random testing procedure, $\text{D}{\scriptstyle\text{RT}}_S$. The procedure is similar to directed random testing for terms. It takes six inputs: (1) a set of operations, (2) a time limit, (3) an oracle, (4) a legality predicate on sequences (Section 5.6), (5) an equivalence relation on sequences (Section 5.7), and (6) an *index pruner p* (described in Section 5.8). $\text{D}{\scriptstyle\text{RT}}_S$ maintains a component set of sequences, initially containing only the empty sequence. It repeatedly generates a sequence from the image of the component set, using the legality and equivalence predicates to discard illegal and equivalent sequences. If a sequence is error-revealing, $\text{D}{\scriptstyle\text{RT}}_S$ outputs the sequence. If no error-revealing sequence is discovered and the time limit expires, the procedure outputs $\perp$ to signal that no error-revealing sequence was discovered.

Let *seqs*(*ops*) be the set of all sequences over operations *ops*. We define the image of a component set of sequences, denoted $I(C, ops)$, as:

$$I(C, ops) = \{s_1 \bullet \ \ldots \ \bullet s_k \bullet op(n_1, \ldots, n_k) \in seqs(ops) : s_i \in C\}$$

```
Procedure $\textsc{Drt}_S$
Input: Operation set $ops$, time limit $t$, oracle $O$, legality predicate $Leg$,
       equivalence relation on sequences $\sim$, index pruner $p$.
Result: An error-revealing sequence, or $\perp$ if none is generated within time limit.
1  $C \leftarrow \{\langle\rangle\}$
2  while time limit $t$ not expired do
3      $img \leftarrow \{s' = s_{pre} \bullet op(n_1, \ldots, n_k) \in I(C, ops) :$
4                  $Leg(op, s_{pre}, n_1, \ldots, n_k) \wedge \forall s_2 \in C : s' \nsim s_2\}$
5      $s =$ rchoose $img$;
6      if $\neg O([\![s]\!])$ then
7          return $s$
8      for $i = 1$ to length of $s$ do
9          $s[i].reusable \leftarrow p([\![s, i]\!])$
10     $C \leftarrow C \cup \{s\}$
11 return $\perp$
```

Figure 5-4: Directed random testing for sequences.

Note that we do not require each name to come from a different sub-sequence. This allows for situations where a single sub-sequence provides more than one argument to the operation. Otherwise, we might miss important sequences from the image. For example, the sequence

`p = Poly(); p.equals(p)`

could not be generated if we insisted that every name come from a different sub-sequence. The sequence `p = Poly(); p.equals(p)` is in the image of any component set that contains `p=Poly()`: supposing C contains `p=Poly()`, then

`p=Poly();`$\langle\rangle$;`p.equals(p)` $\in I(C, ops)$.

The role of the empty sequence is to satisfy the requirement in the definition that there be k input sequences for an operation with k input arguments. The component set is always initialized to contain the empty sequence, to allow compositions like the above.

5.6 Pruning Illegal Sequences

Like Drt, Drt$_S$ takes as input a legality predicate *Leg*, but the predicate takes slightly different arguments. The legality predicate for sequences takes as input an operation *op*, a sequence s, and a vector of names $\langle n_1, \ldots, n_k\rangle$ from the sequence s. The application $Leg(op, s, n_1, \ldots, n_k)$ returns *true* if the values specified by n_1 through n_k in s represent legal arguments to *op*. For example, letting *op* be the `Rat(int,int)` constructor, s be the sequence `i=0; j=1`, and *Leg* be a predicate that reflects the preconditions for `Rat(int,int)`, then $Leg(op, s, \texttt{i}, \texttt{j}) = true$ because `(0,1)` are legal inputs to the constructor, and $Leg(op, s, \texttt{i}, \texttt{j}) = false$ because `(1,0)` are illegal inputs to the constructor.

5.7 Pruning Equivalent Sequences

The notion of equivalence for sequences is analogous to equivalence for terms. An equivalence relation should be *safe*, i.e. not cause DRT_S from discovering an error. It should also be *efficient* (i.e. coarse), to allow for the greatest possible number of discards. The definitions from Section 4.2 carry over to sequences without modification other than changing "terms" to "sequences." An equivalence relation on sequences is **oracle-consistent** if $s \sim s'$ implies $O([\![s]\!]) = O([\![s']\!])$. An **execution** is a run of DRT_S, and is represented as a series of sequences corresponding to the sequences that were added to the component set during the execution. The **scope** of an execution is the set of sequences that can be generated across all possible extensions of the execution. An equivalence relation on sequences is **scope-preserving** if for every execution e and sequence s, there is a sequence $s' \in scope(e)$ where $s' \sim s$. Finally, the crucial definition, safety, is also analogous:

Definition (safe equivalence for sequences). The relation $\sim$ is *safe* iff for any execution e of DRT_S, if there is some sequence $s \in seqs(ops)$ such that $O([\![s]\!]) = false$, then there is a sequence $s' \in scope(e)$ such that $O([\![s']\!]) = false$.

Chapter 4 presented *compositionality*, a property that can be used together with oracle consistency to show that an equivalence relation is safe. The basic idea is the same: using equivalent inputs to an operation results in equivalent output. The definition is slightly more complicated because it must account for names.

Definition (compositional equivalence for sequences). We say that $\sim$ is *compositional* iff for every operation $opname : r_1 \times \ldots \times r_k \to r_{out}$ in *ops*, if $s_1 \sim s_2$ and $s_1 \bullet op(n_1, \ldots, n_k)$ is a valid sequence, then there are names $m_1, \ldots, m_k$ in s_2 such that

$$s_2 \bullet op(m_1, \ldots, m_k) \sim s_1 \bullet op(n_1, \ldots, n_k).$$

This recast definition of compositionality yields a lemma connecting compositionality with scope preservation. The proof follows the same structure as the proof of the lemma 4.2 in Section 4.2.2.

Lemma 5.1. A compositional equivalence relation is scope-preserving.

Proof. Suppose $\sim$ is compositional. We start by proving that for any sequence s and execution e, we can extend e to an execution e' that contains a sequence s' equivalent to s. Finally, since $scope(e)$ includes all the sequences in e', it follows that $s' \in scope(e)$.

Now we prove that for any sequence s and execution e, we can extend e to an execution e' that contains a sequence equivalent to s. The proof is by induction on the length of s. If s is the empty sequence we are done, because every execution contains the empty sequence. Now (induction step), suppose that for all sequences of length less than N, we can extend an execution to one containing an equivalent sequence. We show that the same is true for any sequence of size N. Let $s = s_1 \bullet op(n_1, \ldots, n_k)$ be such a sequence, and e be an arbitrary execution. Using the induction hypothesis, we can extend e to an execution e_1 containing a sequence $s_2 \sim s_1$. By compositionality there is some $m_1, \ldots, m_k$ such that $s = s_1 \bullet op(n_1, \ldots, n_k) \sim s_2 \bullet op(m_1, \ldots, m_k)$. Extending e_1 to contain $s_2 \bullet op(m_1, \ldots, m_k)$, we get an execution that contains a sequence equivalent to s. ■

The possible effects of mutation by an operation are obscured in the proof, because the definition of compositionality already accounts for mutation. Notice that a relation is compositional if, assuming two *arbitrary* equivalent states s_1 and s_2, applying an operation to the states results in equivalent states. The operation could mutate values other than those used directly as input to the operation, but for compositionality to hold in this case, the resulting sequences $s_1 \bullet op(n_1, \ldots, n_k)$ and $s_2 \bullet op(m_1, \ldots, m_k)$ would have to be equivalent, meaning that any significant side effects (e.g. those affecting the result of the oracle) on one would have to be also present in the other. In other words, the lack of mention of mutation in the proof of Lemma 5.1 is due to the fact that mutation is already accounted for in the definition of compositionality.

To show that an equivalence relation over value tuples is safe, we can show that it is compositional and error-consistent, under the new definitions of these concepts. The notion of compositionality for an equivalence relation on sequences is less attractive (and may be more difficult to establish) than compositionality for terms, because it requires reasoning reasoning about the operation's inputs, output, *and* its effect on an *arbitrary* state. In contrast, establishing compositionality for terms requires reasoning only about the operation immediate inputs and output. In going from a setting where operations are functions mapping input values to a single output value without side effects, to a setting where side effects are possible on other values, we gain expressive power, but lose some simplicity in reasoning about equivalence.

5.8 Heuristic Pruning for Sequences

A key idea in directed random testing (and crucial for obtaining a scalable, automated implementation) is to approximate a legality predicate and an equivalence relation based on the runtime behavior of the software under test. A heuristic strategy for determining sequence legality can be essentially the same as for term legality: execute a new sequence, and if the execution completes normally, consider the sequence to be legal. If the execution does not complete normally (e.g., throws an exception), consider the sequence to be illegal.

A heuristic pruning strategy for equivalence is more difficult to translate effectively from the world of terms to that of sequences. Recall the "`equals`-based" strategy for terms described in from Section 4.4: two terms t_1 and t_2 are equivalent if $[\![t_1]\!]$`.equals(`$[\![t_2]\!]$`)` returns `true`. Since each term evaluates to *one value*, and since the `equals` method takes two values as input and returns a boolean, `equals` is a good match (in terms of input and output) for heuristically determining term equivalence.

A natural way to lift the strategy to operation sequences, which produce a *sequence of values*, is element-wise: two sequences s_1 and s_2 are equivalent if they are element-wise equivalent, i.e. if $[\![s_1]\!] = \langle u_1, \ldots, u_k\rangle$, $[\![s_2]\!] = \langle v_1, \ldots, v_k\rangle$, and u_i`.equals(`v_i`)` returns `true` for $1 \leq i \leq k$. Unfortunately, two problems arise when using this `equals`-based strategy on sequences:

1. **Equivalent sequences with important differences.** An `equals`-based approach may consider two sequences equivalent despite important differences arising from relationships *between* elements in the sequence that are ignored when determining equivalence in an element-wise way. An example is shared state: different elements in a sequence may share state that could be important in revealing an error. For example, consider the two sequences

   ```
   s1=PStack();s2=PStack()
   ```

   and

   ```
   s1=PStack();s2=s1.copy().
   ```

   On an element-wise basis, both sequences are equivalent: the first evaluates to a state $\langle s_1, s_2\rangle$ with two empty stacks, as does the second (let its state be $\langle s_3, s_4\rangle$), and using the `PStack.equals` method (which determines two stacks to be equal if they contain the same elements), both s_1`.equals(`s_3`)` and s_2`.equals(`s_4`)` evaluate to `true`. However, the two sequences yield objects with important structural differences: The first stack pair consists of two stacks with separate underlying arrays, while the second stack pair shares the same underlying array, because `copy` (erroneously) assigns the receiver's array to the stack copy. Discarding the second one based on element-wise `equals` can lead the generator to discard the second sequence (which can reveal the `copy` error) if it is generated after the first one.

   Any strategy that determines equivalence in an element-wise way can suffer from the same problem. For example, using other observer methods (e.g. `size()`, `toString()`) to heuristically equivalence among elements from two sequences would suffer the same problem as using `equals`. Unfortunately, observers and equality methods present in code that can be used to perform heuristic pruning are typically designed to operation over single values, or pairs of values, which is a good match when dealing with terms (which evaluate to a single value), but presents the problem that the methods may not be used to detect important relationships *between* values in a sequence.

2. **Non-equivalent sequences with element-wise similarities.** A second problem with using element-wise `equals` computations as a basis for heuristic pruning is that it can introduce a significant amount of redundancy in the component set, even if the component set contains no equivalent sequences.

1	2	3

`Rat r = Rat(2,3);`	`Rat r2 = Rat(1,1);`	`Rat r = Rat(2,3);`
		`Mono m = Mono(r,1);`

4	5	6

`Rat r = Rat(2,3);`
`Mono m = Mono(r,1);`
`Poly p = Poly(m);`
	`PStack s = PStack();`	`Rat r = Rat(2,3);`
		`Mono m = Mono(r,1);`
		`Poly p = Poly(m);`
		`PStack s = PStack();`
		`boolean b = s.push(p);`

Figure 5-5: Six sequences representing an example component set. The sequences are not equivalent, but together they introduce redundancy in the component set. For example, There are four sequence indices that yield the rational number 2/3 (sequences 1, 3, 4 and 6), but only one that yields the rational number 1 (sequence 2).

More specifically, while two sequences may be non-equivalent, they may nevertheless contain many redundant *elements*, and this can result repeated testing of the operations with the same input values. This is best seen with an example. Suppose the first steps in a run of Drt$_S$ results in the sequences shown in Figure 5-5 (in the order shown). Using the element-wise notion of equivalence we just discussed (or even one that allows for equivalent values to be at different indices), all six sequences are non-equivalent; most create collections of values different in both the number of values and their sort. Despite being non-equivalent, the sequences introduce a significant amount of redundancy in the component set, at the *element* level. For example, four elements across all sequences yield the rational value "2/3," but only one element yields the rational value "1." Similarly, three indices across all sequences yield monomials (in sequences 3, 4 and 6), but all three actually yield the same monomial. Element redundancy can cause a directed random testing tool to repeatedly execute operations using the same input values, or distort the distribution towards some values and away from others. For example, when running an early version of Randoop (that did not account for this problem) on object-oriented libraries of containers classes, many of the sequences created empty containers, which are easier to generate than non-empty ones. This caused Randoop to repeatedly test methods on empty containers, wasting time that could have been spent executing methods on more diverse container states.

Both problems can decrease the effectiveness of a directed random test generator, the first by preventing the generation of important test scenarios, and the second by reducing the efficiency of the generator. We address the second problem in this thesis (but not the first). The next section describes *index pruning*, a technique that addresses the problem of redundant elements among a set of sequences. Index pruning attacks this problem by allowing a more fine-grained notion of pruning based not on discarding an entire sequence, but preventing certain elements in a sequence from being reused when creating a new sequence.

Index Pruning

The goal of index pruning is to reduce the number of redundant values present in a component set. As we discussed above, discarding equivalent sequences may not be enough to avoid redundancy in the component set, because a set of non-equivalent sequences can nevertheless contain many redundant elements. The idea is to mitigate this redundancy by (1) detecting elements in a sequence that are redundant (with respect to `equals`) because they can be produced by another sequence already in the component set, and (2) preventing these elements from being reused when generating new sequences. Index pruning is a way to realize, in the context of sequences, the original goal of `equals`-based heuristic pruning: avoiding testing an operation repeatedly on equivalent inputs. We describe index pruning in steps, by describing the three extensions to directed random testing involved: the first extends the definition of a sequence, the second incorporates a pruning step when processing a sequence, and the third modifies the generation of a new sequence to be aware of redundant elements.

- **Reusable element flags.** We modify slightly the definition of an operation sequence, and associate with each element a boolean flag indicating whether the given index can be used when extending the sequence. We can view a sequence as a generator of values; the boolean flags indicate which values the sequence makes available for use in new test inputs. The goal is to prevent multiple sequences from making equivalent values available. Formally, we redefine an operation sequence so that every element is a 4-tuple (operation, input names, output name, and boolean flag). To denote that index i in sequence s has its flag set to *true*, we will write $s[i].reusable = true$. The flags essentially restrict the space of sequences that are generated: the generator never creates a new sequence that uses the given element as input to a new operation.

- **Index pruning procedure.** We augment the directed random testing technique with an *index pruner*, a procedure sets the reusable element flags of a new sequence based on whether an element yields a new value. Throughout the execution of directed random testing, the index pruner maintains an *value set* (not to be confused with the component set), containing every distinct value that has been created across the execution of all sequences, since the start of the generation process.

  $\text{D}\textsc{rt}_S$ invokes the index pruner (lines 8–9 in Figure 5-4) after a new sequence s is generated, and before it is added to the component set. For each element $[\![s, i]\!]$ of the evaluated sequence, $\text{D}\textsc{rt}_S$ queries the index pruner to determine if the reusable flag at i should be set to *true* (the value is new) or *false* (the value is redundant). The pruner determines this by comparing the given value with

1
`T Rat r = Rat(2,3);`
	2
	`T Rat r2 = Rat(1,1);`
		3
		`F Rat r = Rat(2,3);`
		`T Mono m = Mono(r,1);`

4
`F Rat r = Rat(2,3);`
`F Mono m = Mono(r,1);`
`T Poly p = Poly(m);`
	5
	`T PStack s = PStack();`
		6
		`F Rat r = Rat(2,3);`
		`F Mono m = Mono(r,1);`
		`F Poly p = Poly(m);`
		`T PStack s = PStack();`
		`T boolean b =s.push(p);`

Figure 5-6: The same sequences from Figure 5-5, showing boolean flag values obtained through index pruning. The pruning predicate used is heuristic, and it relies on the `equals` method to determine if a value has been created before.

all the values in its value set: if there is no value v in the value set such that v`.equals(`$[\![s,i]\!]$`)` returns `true`, the pruner determines the value to be new, but if for some value in the value set the `equals` method returns `false`, the pruner determines the value to be redundant. (For greater efficiency, the pruner uses a hash set to store values.)

- **Index-aware generation.** The flags affect the generation process as follows: the generator never creates a new sequence that uses a non-reusable element as input to a new operation. We can state this in terms of the image: to do index pruning, we redefine the image of a component set (lines 3–4) to exclude sequences that use any names corresponding to non-reusable indices. For example, consider a component set with two sequences $C = \{$ `r1=Rat(2,3),    r2=Rat(2,3);m=Mono(r2,1)` $\}$ and suppose that the first element of the second sequence shown (the element associated with name `r2`) has been flagged as non-reusable. Then, the sequence

  `r2=Rat(2,3);m=Mono(r2,1);m2=Mono(r2,2)`

  will not be generated, because it uses the element as input to a new operation. (Note that the given monomial can still be generated, by extending the first sequence (which provides an element equivalent to that at `m2`), yielding the sequence `r1=Rat(2,3);m2=Mono(r2,2)`. (In the case of the polynomial classes, index pruning does not hinder the generation of new polynomials; in general, index pruning, like heuristic pruning, has the potential of preventing the generation of useful input values.) To illustrate how index pruning mitigates the problem of redundant elements, Figure 5-6 shows the example execution from Figure 5-5, but also shows the boolean flags of the sequence indices, with index pruning based on the heuristic strategy using `equals`. As Figure 5-6 shows, index pruning in this case solves the repetitive-value problem. For example, the rational value 2/3 has the same number of reusable indices (i.e. one) as the value 1/1, and the repeated monomial value has a single reusable index.

The general concept of preventing certain elements of a sequence from being reused when generating longer sequences can be applied in other useful ways. For example, if an operation returns `null`, Randoop makes the element non-reusable, based on our experience that using `null` values as input to methods often result in arbitrary, but not error-revealing, behavior.

Chapter 6

Post-Processing Techniques

This chapter presents two techniques that post-process the test inputs generated by directed random testing (or any technique that generates operation sequences), to make the output more useful to a test engineer. The first technique, *replacement-based simplification*, aims to help the engineer isolate the root cause of a failing test input results in a failure, and attempts to remove operations from the sequence, or to replace a set of related operations in the sequence with a smaller set, while preserving the failing behavior of the sequence. The result of the technique is a smaller, simplified version of the original sequence that makes it easier to isolate the cause of failure. Simplification is particularly important for directed random testing, which tends to produce long sequences that can be difficult to understand.

The second technique, *branch-directed generation*, aims to increase the code coverage achieved by the set of generated inputs. The technique takes as input a collection of operation sequences, and modifies the sequences with the goal of driving execution through new, uncovered branches in the software under test.

While they have different goals, both techniques share a *modification-based* approach: they modify existing test inputs in order to yield new, more useful inputs e.g. inputs that are smaller and easier to debug, or inputs that cover new branches in the software under test.

6.1 Replacement-Based Simplification

When a test case fails, an engineer must study the test case to find the cause of the failure and fix the underlying error. Directed random testing—and other random testing approaches—tends to generate long failing test cases that include many operations irrelevant to the failure, and which can make it difficult to isolate the failure cause. For example, Figure 6-1 (left) shows an example of a typical test case generated by RANDOOP, our tool implementing directed random testing. The test input shown is a slightly-modified version of a test input generated by the tool when testing the Java JDK. (We mainly removed operations to make the example fit in the page.) The test input contains 32 operations, and reveals an error in method `Collections.unmodifiableSet` that causes it to return an object (`var30`) that fails original test input

```
long[] var1 = new long[]  1L, 10L, 1L;
Vector var2 = new Vector();
PriorityQueue var3 = new PriorityQueue();
Object var4 = var3.peek();
boolean var5 = var2.retainAll(var3);
long[] var6 = new long[]  ;
Arrays.fill(var6, 10L);
IdentityHashMap var7 = new IdentityHashMap(100);
Object var8 = var7.remove(var7);
Set var9 = var7.keySet();
Vector var10 = new Vector(var9);
Vector var11 = new Vector();
IdentityHashMap var12 = new IdentityHashMap(100);
String var13 = var12.toString();
Vector var14 = new Vector();
IdentityHashMap var15 = new IdentityHashMap(100);
char[] var16 = new char[]  ' ', '#', ' ';
int var17 = Arrays.binarySearch(var16, 'a');
Object var18 = var15.remove(var16);
boolean var19 = var14.equals(var15);
boolean var20 = var10.addAll(0, var14);
Collections.replaceAll(var11, var17, var14);
int var21 = Arrays.binarySearch(var1, var17);
Comparator var22 = Collections.reverseOrder();
TreeSet var23 = new TreeSet(var22);
boolean var24 = var23.isEmpty();
Object var25 = var23.clone();
Object[] var26 = new Object []  var21 ;
List var27 = Arrays.asList(var26);
ArrayList var28 = new ArrayList(var27);
boolean var29 = var23.add(var28);
Set var30 = Collections.unmodifiableSet(var23);
assert var30.equals(var30); //fails at runtime
```

removal-minimal input

```
long[] var1 = new long[]  1L, 10L, 1L;
char[] var16 = new char[]  ' ', '#', ' ';
int var17 = Arrays.binarySearch(var16, 'a');
int var21 = Arrays.binarySearch(var1, var17);
Comparator var22 = Collections.reverseOrder();
TreeSet var23 = new TreeSet(var22);
Object[] var26 = new Object []  var21 ;
List var27 = Arrays.asList(var26);
ArrayList var28 = new ArrayList(var27);
boolean var29 = var23.add(var28);
Set var30 = Collections.unmodifiableSet(var23);
assert var30.equals(var30); //fails at runtime
```

after replacement-based simplification

```
TreeSet var1 = new TreeSet();
Object var2 = new Object();
var1.add(var2);
Set var30 = Collections.unmodifiableSet(var1);
assert var30.equals(var30); //fails at runtime
```

Figure 6-1: Three failing test inputs all of which reveal the same error in the JDK method `unmodifiableSet`, namely a failure of reflexivity of `equals` fails for the set denoted by variable `var30` (last method call).

the basic reflexivity of equality contract expected from all Java objects. The same error can be revealed by much shorter sequences of operations, for instance the two sequences on the right-hand side of the figure.

Two previous approaches to test input simplification are delta debugging [99] and dynamic slicing [2]. Delta debugging is a general minimization procedure that isolates failure-inducing program input by systematically removing portions of the input and checking whether the failure still occurs on the smaller input, until a minimal input is reached. Dynamic slicing discards program statements that do not contribute to a particular computation. Delta debugging and dynamic slicing can identify operations that can be safely removed from a sequence without affecting its error-revealing behavior.

Both delta debugging and dynamic slicing are fundamentally limited to *removing* operations from a failing sequence. However, a semantically extraneous operation may be impossible to remove, if its removal causes the sequence to be syntactically invalid. For example, recall the error in method `Poly.add` (Figure 2-1) that causes the method to incorrectly add a polynomial p and a monomial m if m is the negation of an element in p (e.g. $p = 2x^3 + 1$ and $m = -2x^3$). The following sequence reveals the error:

```
1. Rat r1  = new Rat(1,1);      // r1: 1
2. Rat r2  = new Rat(2,1);      // r2: 2
3. Rat r3  = new Rat(-2,1);     // r3: -2
4. Mono m1 = new Mono(r1,2);    // m1: x^2
5. Mono m2 = new Mono(r2,3);    // m2: 2x^3
6. Mono m3 = new Mono(r3,3);    // m3: -2x^3
7. Poly p1 = new Poly(m1);      // p1: x^2
8. Poly p2 = p1.add(m2);        // p2: 2x^3 + x^2
9. Poly p3 = p2.add(m3);        // p3: 0x^3 + x^2  (breaks rep. invariant)
```

The sequence is *removal-minimal*, meaning that no operations (i.e. lines) can be removed from the sequence without making it syntactically invalid, and no longer error-revealing. For example, removing line `1`, which defines variable `r1`, would cause line `4`'s call `new Mono(r1,2)` to refer to a non-existent variable, and the resulting sequence would not be well-typed.

We can simplify the sequence further if we *replace* some of the operations. For example, replacing operations `1`, `4` and `7` by a new operation (indexed `R` below), we obtain a smaller error-revealing sequence:

```
2. Rat r2  = new Rat(2,1);      // r2: 2
3. Rat r3  = new Rat(-2,1);     // r3: -2
5. Mono m2 = new Mono(r2,3);    // m2: 2x^3
6. Mono m3 = new Mono(r3,3);    // m3: -2x^3
R. Poly p2 = new Poly(m2);      // p2: 2x^3
9. Poly p3 = p2.add(m3);        // p3: 0x^3        (breaks rep. invariant)
```

Replacement-based simplification is a novel test simplification technique that removes *and* replaces operations in a failing sequence. The technique can be more effective in reducing the size of a sequence than previous approaches to test case simplification based exclusively on removing operations. For example, the top-right sequence in Figure 6-1 is the removal-minimal version of the original sequence (on the left portion of the figure), representing the best-possible result that can be obtained by delta debugging or slicing. The bottom-right sequence in the figure shows the result of applying replacement-based simplification, which both removes and replaces operations. The resulting sequence has only 4 operations, which is in fact the smallest number of operations that reveals this particular error. (In general, the technique may not always produce a minimal sequence.)

To perform replacements, the technique requires a *replacement set* of sequences that it can splice into the failing sequence when attempting to simplify it. This makes the technique a natural step to follow directed random testing: the latter can supply as a replacement set the component set of sequences generated during the run of the generator.

Simplification Technique

A component-based simplifier takes three inputs: (1) a test oracle O, (2) a *target sequence* s that fails the oracle, and (3) a set of sequence S_R, which we call the "replacement set." We assume that the oracle is a distinguishing oracle (Section 4.2.3), i.e. a function that maps a runtime value to a failure class. In other words, O is a function $O : \mathit{Vals} \rightarrow \mathit{ErrorClasses}$, where *Vals* is the universe of runtime values. When the simplification technique modifies the failing sequence s (obtaining a new sequence s'), it uses the oracle to check whether the modification preserves the failing behavior. It does this check using the oracle: if $O([\![s']\!]) = O([\![s]\!])$, the simplifier determines that the failing behavior is preserved by the modified sequence s'. The simplifier outputs *either* the original target sequence (if it was not able to simplify the sequence), *or* a new sequence guaranteed to be smaller than the target sequence, and to exhibit the same failing behavior.

The simplifier works in two phases: the first is a *removal* phase, and the second is a *replacement* phase. The removal phase consists in using a removal-based technique to remove as many operations as possible from the input sequence. The simplifier can use a technique like delta debugging or dynamic slicing to perform this phase. Our implementation uses a simpler removal technique that proves to be quite effective in practice. The technique attempts to remove one operation at a time, processing operations in the order that they appear in the sequence, but starting from the last operation and working towards earlier operations. The simplifier only removes an operation from the sequence if the sequence after the removal is syntactically valid: a sequence is *syntactically valid* if every use of a variable in the sequence is preceded by an operation that produces the variable, and all variable uses are sort-consistent. After removing an operation, the simplifier executes the sequence and evaluates the oracle to check if the failing behavior is preserved. If not, it reinstates the operation. An example of the sequence that this technique yields is the removal-minimal sequence from Figure 6-1.

The second phase is the replacement phase, and is the core of the simplification technique. Like the removal phase, the simplifier processes the operations in the sequence in reverse, starting from the last operation, and stopping after it is done processing the first operation. The simplifier maintains an index *idx* that denotes the operation being currently processed. To process one operation, the simplifier performs the following steps.

a. **Remove operation.** The first step removes the operation at index *idx*, and replaces it with a placeholder symbol "□", to be filled in a later step. For example, let the sequence to be simplified be the polynomial sequence we discussed earlier in the section:

```
1. Rat r1  = new Rat(1,1);
2. Rat r2  = new Rat(2,1);
3. Rat r3  = new Rat(-2,1);
4. Mono m1 = new Mono(r1,2);
5. Mono m2 = new Mono(r2,3);
6. Mono m3 = new Mono(r3,3);
7. Poly p1 = new Poly(m1);
8. Poly p2 = p1.add(m2);
9. Poly p3 = p2.add(m3);    // breaks rep. invariant
```

removing operation 7 (step a)	removing related operations (step b)
`1. Rat r1  = new Rat(1,1);`	
`2. Rat r2  = new Rat(2,1);`	
`3. Rat r3  = new Rat(-2,1);`	`Rat r3  = new Rat(-2,1);`
`4. Mono m1 = new Mono(r1,2);`	
`5. Mono m2 = new Mono(r2,3);`	
`6. Mono m3 = new Mono(r3,3);`	`Mono m3 = new Mono(r3,3);`
`7. Poly p1 = new Poly(m1);`	
`8. Poly p2 = □;`	`Poly p2 = □;`
`9. Poly p3 = p2.add(m3);`	`Poly p3 = p2.add(m3);`

Two examples of step (d)

successful replacement	unsuccessful replacement
`// new operations`	
`Rat r  = new Rat(2,1);`	`//new operations`
`Mono m = new Mono(r,3);`	Poly $\boxed{p}$ = new Poly();
Poly $\boxed{p}$ = new Poly(m);	
`// old operations`	`// old operations`
`Rat r3  = new Rat(-2,1);`	`Rat r3  = new Rat(-2,1);`
`Mono m3 = new Mono(r3,3);`	`Mono m3 = new Mono(r3,3);`
~~Poly p2 = □;~~	~~Poly p2 = □;~~
Poly p3 = $\boxed{p}$.add(m3);	Poly p3 = $\boxed{p}$.add(m3);

Figure 6-2: Example of particular steps in the simplification technique.

Figure 6-2 (left) shows the sequence after this step performed, assuming *idx* = 8. We call the variable associated with the placeholder operation (`p2` in the example) the *placeholder variable*. For the purpose of explanation, we consider a sequence containing a placeholder operation to be syntactically valid.

b. **Remove related operations.** After the operation has been replaced with □, it may be possible to remove additional operations in the sequence and still preserve syntactic validity, because operations providing inputs to the (removed) operation at *idx* are no longer necessary. This step removes all operations from the sequence that can be removed while preserving syntactic validity. Figure 6-2 (top right) shows the example sequence after this step, which results in 5 additional operation removals.

c. **Select replacement candidates.** Let N be the number of operations removed in steps (a) and (b). The goal of the remaining steps is to find a sub-sequence (to be inserted in the target sequence) of length less than N that provides an operation to replace the placeholder, while preserving the failing behavior. If such a sub-sequence is found, the resulting sequence will be shorter in length than the original, constituting progress.

To find such a sub-sequence, the simplifier selects a set of *candidate sub-sequences* from the replacement set S_R. Candidate sub-sequences must have length less than N and assign a variable of a type compatible with the *uses* of the placeholder variable. In other words, a candidate sub-sequence must declare a variable that, when substituted for the placeholder variable in the target sequence, yields a syntactically-valid sequence. For example, a candidate sub-sequence for the example target sequence in Figure 6-2 would be the following sequence that declares a variable `p` of sort `Poly`, because `p` can replace the placeholder variable `p2`.

```
Rat r  = new Rat(1,1);
Mono m  = new Mono(r4,3);
Poly p = new Poly(m4);
```

If the simplifier finds no candidate sub-sequences in the replacement set, it decrements idx to $idx - 1$ and processes the next operation.

d. **Attempt replacements.** At this point, the simplifier has a collection of candidate sub-sequences. Next, the simplifier iterates through every candidate sub-sequence s_r in order of increasing length. Since s_r is a candidate sub-sequence, it declares one or more *replacement variables* of a sort compatible with all uses of the placeholder variable. The simplifier iterates through the candidate variables in s_r.

   Let s be the target sequence, v_p be the placeholder variable in s, s_r be the candidate sub-sequence, and v_r be the replacement variable in s_r. The simplifier (1) creates the new sequence $s_r \bullet s$ consisting of the target sequence prepended with the candidate sub-sequence, (2) replaces every occurrence of the placeholder variable v_p with the replacement variable, v_r, and (3) removes the placeholder operation. Figure 6-2 (bottom) shows two examples of this step; the first example is for the candidate sub-sequence

```
Rat r  = new Rat(1,1); Mono m  = new Mono(r,3); Poly p = new Poly(m)
```

   and replacement variable `p`, and the second example is for the candidate sub-sequence

```
Poly p = new Poly().
```

   and replacement variable `p`. After performing a replacement, the simplifier re-executes the modified sequence and uses the oracle to determine if the failure is preserved. If not, the simplifier undoes the modification. The two replacements in Figure 6-2 exemplify a successful replacement (failure is preserved) and an unsuccessful one.

   The simplifier iterates through all candidate sequence/replacement variable pairs. It does this until it either succeeds in replacing the operation at index idx (and its related operations) with a smaller set of operations that preserves the failure, or until no more pairs remain.

Steps in the replacement-based simplification of a JDK failing sequence.

~~`long[] v1 = new long[]1L,10L,1L;`~~
~~`char[] v16 = new char[]' ','#',' ';`~~
~~`int v17 = Ar.binarySearch(v16,'a');`~~
~~`int v21 = Ar.binarySearch(v1,v17);`~~
`Comparator v22 = reverseOrder();`
`TreeSet v23 = new TreeSet(v22);`
~~`Object[] v26 = new Object[] v21 ;`~~
~~`List v27 = asList(v26);`~~
`idx  ArList v28 = □` ~~`new ArList(v27);`~~
`boolean v29 = v23.add(v28);`
`Set v30 = unmodSet(v23);`
	$\boxed{\texttt{Object o = new Object();}}$
	
	`Comparator v22 = reverseOrder();`
	`TreeSet v23 = new TreeSet(v22);`
	
	`boolean v29 = v23.add(`$\boxed{\texttt{o}}$`);`
	`Set v30 = unmodSet(v23);`

`Object o = new Object();`
 	
~~`Comparator v22 = reverseOrder();`~~
`idx  TreeSet v23 = □` ~~`new TreeSet(v22);`~~
 	
`Object v28 = o;`
`boolean v29 = v23.add(o);`
`Set v30 = unmodSet(v23);`
	`Object o = new Object();`
	$\boxed{\texttt{TreeSet t = new TreeSet();}}$
	
	`boolean v29 = `$\boxed{\texttt{t}}$`.add(v28);`
	`Set v30 = unmodSet(`$\boxed{\texttt{t}}$`);`

Figure 6-3: Steps in the replacement phase for the sequence from Figure 6-1. For brevity, only steps resulting in a successful replacement are shown. A line preceded by `idx` indicates the index of the current variable whose value is being replaced, crossed-out lines represent the statements removed, and boxed lines represent replacement sub-sequences and variables. We abbreviate JDK class and method names for space reasons.

e. **Update index.** Finally, the simplifier updates the index *idx* in order to process the next operation. If the simplifier failed to perform a replacement in step (d), it sets *idx* to $idx - 1$. If it succeeded in performing a replacement, it sets *idx* to $idx - N + |s_r|$, where N is the number of operations removed in steps (a) and (b), and $|s_r|$ is the length of the replacement sequence. For example, in the bottom-left example in Figure 6-2, the new index after the successful replacement will be $8 - 6 + 3 = 5$, which points to the next operation to be processed, i.e. `Mono m3 = new Mono(r3,3)`.

At the end of the above process, the simplifier returns the modified sequence.

Figure 6-3 shows the simplification procedure applied to the removal-minimal sequence from Figure 6-1. For brevity, we only show the two steps that resulted in a successful replacement. The top row shows the first replacement, which replaces the operation associated with `v28` variable, plus five related operations, with the simpler sub-sequence `o=new Object()`. The bottom rows show the second replacement, which replaces the operation associated with v23, plus one related operation, with the simpler sub-sequence t = new TreeSet(). Two examples of unsuccessful replacements that preceded the successful ones: (1) when replacing v29, the simplifier first attempted to use as replacement the sequence int i = 1; (2) when replacing v23, the simplifier first attempted to use as replacement the sequence t = Collections.emptySet().

The simplifier iterates through a set of candidate sub-sequences (step d) in order of increasing length, to guarantee that if there is a successful replacement, it will be minimal in length. Among all sub-sequences with the same length, our implementation of the technique additionally uses a partial order among sub-sequence/variable pairs where $\langle s_1, v_1 \rangle < \langle s_2, v_2 \rangle$ iff v_1's type is a super-type of v_2's type. The simplifier iterates through pairs in arbitrary order as long as it respects this partial order. The strategy causes replacement variables with more general types to be attempted earlier. The rationale for the strategy is that given the choice between two replacements of the same length, the one that yields a value with a more general type can make it easier to isolate the cause of a failure, because general types tend to have a smaller set of behaviors. The most compelling example of this is the sub-sequence o=new Object() which, due to the partial ordering, is usually one of the first sub-sequences tried, and can lead to significantly simpler sequences, as in the simplified sequence from Figure 6-3.

6.2 Branch-Directed Generation

Branch-directed generation is a test generation technique whose goal is to generate test inputs that cover branches left uncovered by a previous input generation technique. A branch-directed generator takes an input set (e.g. one produced by a previous run of directed random testing) and outputs a new input set containing inputs that cover branches left uncovered by the old set. Its generation strategy is to select inputs from the old set that reach but do not follow a particular branch in the code, and to make small, targeted modifications to the inputs, guided by a dynamic data flow analysis of values that impact the branch condition.

A key assumption of the technique is that small, simple modifications to an input can be sufficient to achieve new code coverage. Unlike dynamic generation techniques that use constraint solvers to exactly solve path constraints and yield a new input that covers a new branch (like DART [39] or concolic testing [86]), branch-directed generation performs simple modifications selected from a fixed set of strategies, e.g. changing the sign of an integer value, or incrementing the value by a constant. Our preliminary evaluation of the technique (Section 8.4) suggests that such modifications can be effective in flipping a branch condition. The rest of this section describes the technique and illustrates it with an example.

Branch-Directed Generation Technique

A branch-directed generator takes as input a set of sequences. The first step in the technique computes the branch coverage in the software under test achieved by set

Suppose the underlined condition is a **frontier condition**: an input set covered the *false* branch, but never covered the *true* branch.

```
public class Poly {
 public Poly add(Mono m) {
  List<Mono> mList = new ArrayList<Mono>();
  for (int i=0 ; i< ms.length ; i++) {
    if (ms[i].exp == m.exp) {
      Rat sum = ms[i].coeff.add(m.coeff);
      Mono msum = new Mono(sum, m.exp);
      mList.add(msum);
      continue;
    }
  ...
  }
 }
}
```

This input is a **witness** for the frontier. Boxed values are **influencing variables**.

```
int i1 = 2;
int i2 = 1;
Rat r1 = new Rat(i1,i2);
int i3 = 2;
Mono m1 = new Mono(r1,i3);
Poly p1 = new Poly(m1);
int i4 = 1;
int i5 = 1;
Rat r2 = new Rat(i4,i5);
int i6 = 1;
Mono m2 = new Mono(r2,i6);
Poly p2 = p1.add(m2);
```

Modifying `i3` yields an input that covers the frontier's *true* branch.

```
int i1 = 2;
int i2 = 1;
Rat r1 = new Rat(i1,i2);
int i3 = 1;
Mono m1 = new Mono(r1,i3);
Poly p1 = new Poly(m1);
int i4 = 1;
int i5 = 1;
Rat r2 = new Rat(i4,i5);
int i6 = 1;
Mono m2 = new Mono(r2,i6);
Poly p2 = p1.add(m2);
```

Figure 6-4: Example artifacts used by a branch-directed generator. The bottom right box shows an input that would be generated by the technique.

of sequences, and classifies branch conditions into *fully-covered conditions* (both *true* and *false* branches are covered), *fully-uncovered conditions* (no branches are covered), and *frontier conditions* (one branch branch is covered, but not the other). It also computes the set of *witness sequences*: sequences whose execution reaches a frontier condition. Figure 6-4 shows an example of a frontier condition (top half, condition is underlined) resulting from a hypothetical set of sequences, and a witness sequence (bottom left) that reaches the frontier condition.

The next step in the technique modifies the witness sequences with the goal of covering uncovered branches in the frontier conditions. The generator focuses on frontier conditions because, as their name suggests, they represent the boundary of coverage: fully-covered conditions can be ignored, and fully-uncovered conditions may be more difficult to cover because there is no sequence that reaches them. Covering uncovered branches in frontier conditions can also lead to new coverage, including new coverage of fully-uncovered conditions. As an example of a modification performed by the technique, Figure 6-4 (bottom right) shows a modification to the witness sequence that covers the missing branch in the condition. The modification (changing the variable's value from `2` to `1`) is one of several that the branch-directed generator attempts, and which we describe below.

To perform modifications, the branch-directed generator iterates through the frontier conditions. For each frontier condition C, it collects all the witness sequences for the condition, and attempts to modify each in turn, until the uncovered branch in C is covered or no witness sequences are left. To modify a witness sequence, the generator performs the following steps.

a. **Collect runtime information.** The generator executes the sequence and collects runtime information that it uses to guide the modifications, including dynamic data flow information and information about values that were compared at runtime. To perform dynamic data collection, we use a modified version of DynCompJ [45], a tool that can track the flow of values during the execution of a Java program. The tracker associates with every runtime value v (primitive or pointer) a tag that uniquely identifies the value. Tags allow the tracker to determine the flow of values stored at different locations, even when the values are identical, e.g. the values stored in variables `i1` and `i3` of the example sequence in Figure 6-4. As the sequence executes, the tracker maintains three pieces of data associated with each tag.

   - **Related tags.** The set of related tags for a tag t, denoted $rel(t)$, is a set of tags representing values that contributed to the computation of t's value. As the sequence executes, the tracker updates these sets to reflect the dynamic data flow occurring during the execution. For example, when a value is first created, the tracker assigns a new tag t to the value, and sets $rel(t)$ to $\{\}$. When an assignment statement like `x=y+z` occurs, the tracker creates a new tag t for the value resulting from the addition, and sets $rel(t)$ to $rel(x) \cup rel(y)$ (where x and y are the operand values of the addition), to reflect the fact that the operands (and, transitively, values related to them) contributed to the result of the addition.
   - **Source variable.** The name of the source variable for tag t, denoted $var(t)$, specifies the variable in the sequence that is associated with a tag, if there is one. For example, when the value associated with variable `i1` in the sequence from Figure 6-4 is created, the tracker creates a new tag t for the value and sets $var(t)$ to `i1`. Not all tags are associated with a sequence variable—for example, there is no association for tags representing intermediate values inside an operation that are never assigned to an externally-visible variable in the sequence.
   - **Compared values.** The *compared-value set* for tag t is the set of all runtime values that the value of t was compared to (using a comparison operation like `==`, `<`, etc.) during the execution of the operations in the sequence.

b. **Derive influencing variables.** An influencing variable is a variable in the sequence whose runtime value contributes to the result of the frontier condition. Figure 6-4 shows a witness sequence for a frontier branch in the `Poly.add` method. It also shows (in boxes) the influencing variables `i3` and `i6`, representing the sequence variables whose runtime values are compared in the frontier branch condition. By discovering influencing variables, the generator can determine variables that affect the outcome of the frontier condition and focus its effort on modifying the values of those variables, with the goal of flipping the condition's result.

   The generator determines the influencing variables of a sequence by inspecting the related tags and source variable data computed in step (a). When execution of the sequence reaches the frontier condition, the tracker inspects the operand values $v_1, \ldots, v_k$ associated with the conditional operation, determines their tags $t_1, \ldots, t_k$, and obtains the set of tags $rel(t_1) \cup \ldots \cup rel(t_k)$ representing those values that flowed to the condition's result. It uses the source variable information to determine the influencing variables, i.e. variables in the sequence associated with the set of tags.

c. **Modify influencing variables.** The set of influencing variables represent variables in the sequence whose value flowed to the result of the frontier condition, making these variables promising candidates for modification. The generator systematically modifies the values of influencing variables with the aim of flipping the result of the branch condition. It loops through a fixed set of modification strategies, described below. For each strategy, it modifies the sequence according to the strategy and re-executes it to check if the uncovered branch of the frontier is covered by the modified sequence. The generator has two kinds of modification strategies: *unary* modifications modify the value of a single influencing variable, and *binary* modifications simultaneously modify the values of two influencing variables. The modification strategies below are the ones we have currently implemented, but they could be extended.

   - **Unary modifications.** For every influencing variable with runtime value v, try the following modifications:
     - set the variable to v' for every element v' in the compared-value set of v.
     - For variables with numeric sort (e.g. `int` or `double` in Java), set the variable to $-v$, 0, and $v + n$ for $n \in \{-1, 1, -10, 10, -100, 100, ...\}$ up to some user-settable limit.
     - For variables with pointer sort, set the variable to `null`.
   - **Binary modifications.**
     - For every pair of influencing variables `v1` and `v2` with compatible numeric sorts (the value of one can be assigned to the other and vice-versa) and runtime values v_1 and v_2 (i.e. `v1` has value v_1 and `v2` has value v_2), set `v1` to v_2 and set `v2` to v_1.
     - For every pair of influencing variables `v1` and `v2` with compatible reference sorts, syntactically replace all *uses* of `v1` in the sequence by `v2`, and

vice-versa. (If the variable to be replaced is used before the replacement variable is defined, the modification is not performed because it would result in an invalid sequence.)

If a modification results in an input that goes through the uncovered branch in C when executed, the generator has succeeded in covering the uncovered branch in the frontier condition. It proceeds to the next frontier condition, collects a set of witnesses for the new condition, and repeats the process for each witness, starting from step (a).

At the end of this process, the generator outputs the set of modified sequences (if any) whose execution covered a branch not covered by the original input set. The output set may cover more branches than those in the frontier conditions: a sequence that covers a previously-uncovered branch b may also cover other branches previously guarded by b.

The bottom boxes in Figure 6-4 show a successful instance of the unary modification strategy that sets the value of an influencing variable (`i3`) to a different value (`1`) found in the variable's compared-value set. Variable `i3`'s compared-value set contains the value `1` because `i3`'s value is compared to `1` in the frontier branch condition of `Poly.add` shown in the figure. Section 8.4 shows more successful modification examples from the application of the technique to the Java JDK.

Chapter 7

Randoop: Directed Random Testing for Java and .NET

The previous chapters have introduced the ideas and techniques that make up the directed random testing approach. To get the key ideas across, we described the techniques at a level of abstraction that omits details specific to a particular programming language or environment. This chapter bridges the gap between the abstract concepts we have developed and a concrete implementation. We now describe RANDOOP, a tool that implements directed random testing for testing object-oriented software at the unit level. We have implemented two versions of the tool, one targeting Java [40, 41] (developed at MIT and publicly available [84]), and the second targeting .NET [74] (used internally at Microsoft). The .NET version of the tool differs from the Java version in a few respects; below, we describe the differences as they arise.

In implementing RANDOOP, our goal was to create a tool that could generate a large volume of test cases in a relatively short amount of time, and that would do so completely automatically (at least in its default usage mode). In other words, we wanted RANDOOP to preserve automation and scalability, characteristics that make random testing tools attractive and easy to adopt. This influenced key decisions in the design of the tool. For example, RANDOOP executes test inputs via reflection in the same process that RANDOOP runs, which while less safe (RANDOOP occasionally crashes due to execution problems with the software under test), is an order of magnitude faster than compiling and executing tests in a separate process. This makes RANDOOP able to generate thousands of test cases in minutes, compared with hundreds using separate compilation of tests. To make RANDOOP fully automatic, we implemented a set of default, general heuristic pruning techniques that work effectively in many cases, even if they are not ideal for every software artifact under test. The user can tailor the tool to specific uses, using a visitor-based extension mechanism that can be used to add new test oracles and pruning heuristics.

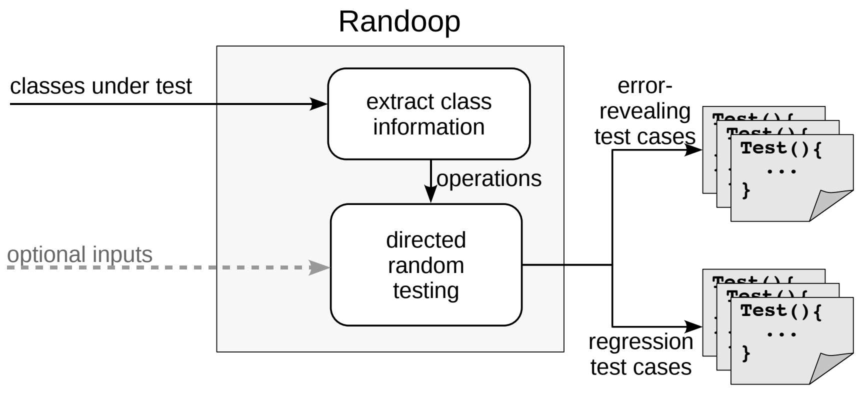

Figure 7-1: High-level overview of Randoop, including its input and output artifacts.

7.1 Object-Oriented Directed Random Testing

Object-oriented programming is a style of programming in which an *object* is a core abstraction that encapsulates *state* and *behavior*. An object stores its state in named variables called *fields* and exposes its behavior via *methods*, which are like functions in other programming languages. A programmer defines the blueprint for an object by defining a *class* declaring the fields and methods that specify the state and behavior for instance objects of the class.

We have already seen examples of Java classes in the examples used to motivate the techniques in previous chapters. As previous chapters show, there is a natural mapping from the abstract concepts of operations, sorts, and values used to describe our techniques to object-oriented artifacts like methods, classes, and objects. Our implementation of directed random testing uses this natural mapping. One aspect of an object-oriented language like Java that is not captured precisely by our abstractions is that of primitive values versus objects; in our treatment, we make no such distinctions, and simply have operations. Our implementation treats primitive values as (0-argument) operations; for example, Randoop treats the Java integer `0` as an operation that takes no inputs and returns the corresponding value.

7.2 Randoop Input and Output

Figure 7-1 shows a high-level view of Randoop's input and output. The only required input to the tool is a set of classes under test. Optional arguments include a time limit (the default is 2 minutes), a random seed (the default is "0"), test oracles in addition (or replacing) the tool's default oracles (Section 7.7), and the desired output format (e.g. JUnit, Java, plain text, etc). Randoop outputs two tests suites. One contains *contract-violating tests* that exhibit scenarios where the code under test leads to the violation of an API contract. For example, Figure 7-2, (reproduced from Chapter 1) shows a contract-violating test that reveals an error in Java's collections framework. The test case shows a violation of the `equals` contract: a set `s1` returned by `unmodi-`

Error-revealing test case

```
// Fails on Sun 1.5.
public static void test1() {
   LinkedList l1 = new LinkedList();
   Object o1 = new Object();
   l1.addFirst(o1);
   TreeSet t1 = new TreeSet(l1);
   Set s1 =
    Collections.unmodifiableSet(t1);
   Assert.assertTrue(s1.equals(s1));
}
```

Regression test case

```
// Passes on Sun 1.5, fails on Sun 1.6
public static void test2() {
   BitSet b = new BitSet();
   Assert.assertEquals(64, b.size());
   b.clone();
   Assert.assertEquals(64, b.size());
}
```

Figure 7-2: Example Randoop output, including a contract-violating test cases (left) and regression test cases (right).

`fiableSet(Set)` returns `false` for `s1.equals(s1)`. This violates the reflexivity of `equals` as specified in Sun's API documentation. This test case actually reveals two errors: one in `equals`, and one in the `TreeSet(Collection)` builder, which fails to throw `ClassCastException` as required by its specification. Randoop uses a default set of contracts (including reflexivity of equality) that the user can extend.

Randoop implements test oracles using *API contracts*, or correctness properties specified at the application programming interface (API) level. Randoop uses two types of contracts. An *object contract* expresses an invariant property that should always hold of an object. A *method contract* expresses a property that must hold after a method is executed. Figure 7-3 (top) shows Randoop's default method and object contracts, most of which we based on the expected behavior Java and .NET classes documented in the API specification of the classes.

The second suite that Randoop outputs contains *regression tests.* which do not violate contracts but instead capture an aspect of the current implementation. For example, the second test in Figure 1-2 shows a regression test that records the behavior of the method `BitSet.size()`. Regression tests can discover inconsistencies between two different versions of the software. For example, the regression test shown reveals an inconsistency between Sun's JDK 1.5 (on which the test was generated, and succeeds by construction) and Sun's JDK 1.6 Beta 2 (on which the test fails). Contract-violating tests and regression tests both have as their purpose to reveal errors: the former reveal potential errors in the current implementation of the classes under test, while the latter may reveal errors in future implementations.

7.3 Heuristic Pruning in Randoop

Randoop does not require as input an oracle, legality predicate, an equivalence relation on sequences, or an index pruning predicate, because it uses default versions of these artifacts. The defaults are extensible; our experience (Chapters 8 and 9) suggests that they are effective to reveal errors in a large set of classes. However, other users of the tool have found it useful to extend them with ad hoc constructs more appropriate to their classes under test. (For an example, see Bacchelli et al.'s

Randoop's Oracle

Method contract	**description**
Exception (Java)	Method throws no `NullPointerException` if no null input parameters.
	method throws no `AssertionError`.
Exception (.NET)	Method throws no `NullReferenceException` if no null input parameters.
	method throws no `IndexOutOfRangeException`
	method throws no `AssertionError`

Object contract	**description**
equals reflexive	`o.equals(o)` returns `true`
equals/null	`o.equals(null)` returns `false`
equals/hashCode	if `o1.equals(o2)`, then `o1.hashCode()=o2.hashCode()`
equals symmetric	if `o1.equals(o2)`, then `o2.equals(o1)`
hashCode	`o.hashCode()` throws no exception
toString	`o.toString()` throws no exception

Randoop's Legality Properties (term is judged illegal if...)

null	Throws `NullPointerException`, and `null` *was* used as input to some call.
exception	Throws an exception not covered by any other cases in this table.
termination	Sequence execution appears non-terminating.

Figure 7-3: Top: default contracts checked by Randoop. Users can extend these with additional contracts, including domain-specific ones. A contract is created programmatically by implementing a public interface. Bottom: Randoop's heuristics for determining legality.

case study of Randoop and other testing tools [10].)

Legality

Figure 7-3 (bottom) shows the heuristic legality properties that Randoop uses to determine if a method call execution was illegal. In summary, a method calls is deemed illegal if it results in an exception not covered by the correctness contracts. The rationale for this is that in Java and .NET, exceptions are often indicative of a problem with the state of the objects and indicate that incorrect inputs were given to the method.

A method that appears not to terminate is also considered to have been an illegal call. Randoop has a time threshold parameter; if a method's execution lasts beyond the threshold, the method execution is terminated and deemed illegal due to possible non-termination. The parameter is user-settable, and defaults to 5 seconds.

Equivalence

Randoop determines two two sequences to be equivalent if they are identical, modulo index names; if a sequence is generated that is equivalent to a previously-generated sequence, Randoop discards it before executing it, and attempts to generate a new sequence. To efficiently check if a sequence is identical to an already-created sequence, the tool defines a hash function over sequences, stores the sequences in a hashtable, and checks a new sequence for membership in the hashtable.

7.4 Index Pruning in Randoop

Randoop uses three index pruning strategies.

Object equality

Randoop uses the `equals` method to determine if an object with the same abstract state has been created by a previously-created sequence. The filter maintains a set *allObjs* (distinct from the component set) of all `equals`-distinct objects that have been created by the algorithm across all sequence executions (the set can include primitive values, which are boxed). For each object *o* created by a new sequence, it sets the sequence index corresponding to that object to *reusable* if there is an object in *allObjs* that is equivalent, using `equals` methods in the software under test to determine object equivalence. To make the comparison more efficient, Randoop implements the *allObjs* set using a hash set.

This filter prunes any object with the same abstract value as a previously-created value, even if their concrete representations differ. This might cause Randoop to miss an error, if method calls on them might behave differently. The heuristic works well in practice but can be disabled or refined by the user. For instance, Randoop has an optional flag that, when set, instead of using `equals` uses reflection to determine whether two objects have the same concrete representation (the same values for all their fields, recursively). A user can also specify, via another command-line argument, more sophisticated or ad hoc computations to determine value equivalence.

Randoop for .NET does not use the above object-equality technique. Instead, it determines at runtime if a method call appears to have mutated the state of its arguments (using the `equals` method), and makes the arguments reusable only if their state is mutated by the call.

Null values

Null dereference exceptions caused by using `null` as an argument are often uninteresting, and usually point to the (possibly intentional) absence of a null check on the arguments. However, when a null dereference exception occurs in the absence of any null value in the input, it often indicates some internal problem with the method.

Note that null arguments at runtime are possible even if a sequence does not explicitly use `null` as an input to a method. Null values are hard to detect statically because the arguments to a method in a sequence themselves are outputs of other sequences. The null filter sets a sequence element's *reusable* flag to *false* iff the corresponding runtime value is `null`, which can happen even if a sequence does not use `null` explicitly.

Large numbers

In our experience using RANDOOP, we noticed that using very large numbers as inputs to methods rarely revealed errors, and often resulted in very long method executions that slowed down the tool. Even though RANDOOP terminates a method if it takes more than 5 seconds, if this behavior happens very frequently, it degrades the overall input generation efficiency of the tool. This filter disables an index if it corresponds to a numeric value that exceeds a value limit specified by the user.

7.5 Assembling a New Sequence

The directed random testing technique described in Chapter 5 "constructs" a new sequence by selecting it at random from the image of the component set. An implementation that actually computed the image of the component set (only to select a single element from it) would spend most its time computing image sets. Instead, RANDOOP incrementally assembles a *single* new sequence from those in the component set, by selecting a method at random and finding sequences in the component set that create values of the input types required by the method.

To simplify the explanation of RANDOOP's sequence assembly process, we define an operator *extend(m, seqs, names)* that takes zero or more sequences and produces a new sequence. The extension operation creates a new sequence by concatenating its input sequences and appending a method call at the end. More formally, the operator takes three inputs:

- ***m*** is a method with formal parameters (including the receiver, if any) of type $T_1, \ldots, T_k$.
- ***seqs*** is a list of sequences.
- ***names*** is a list of names $n_1 : T_1, \ldots, n_k : T_k$. Each name corresponds to the return value of a method call appearing in a sequence from *seqs*.

The result of *extend(m, seqs, names)* is a new sequence that is the concatenation of the input sequences *seqs* in the order that they appear, followed by the method call $m(n_1, \ldots, n_k)$. Figure 7-4 shows three examples of applying the operator. Both reuse of a value (as illustrated in the first example) and use of distinct duplicate sequences (as illustrated in the third example) are possible.

Sequence creation first selects a method $m(T_1 \ldots T_k)$ at random among the methods under test. Next, it tries to apply the extension operator to m. Recall that the operator also requires a list of sequences and a list of names. RANDOOP incrementally builds a list of sequences *seqs* and a list of names *names*. At each step, it adds a name to *names*, and potentially also a sequence to *seqs*. For each input argument of type T_i, it adds a name to *names*, and potentially also a sequence to *seqs*, by selecting one of the three possible actions at random:

- Use a name (with reusable index) from a sequence that is already in *seqs*.

- Select a new sequence from the component set that has a name (with reusable index) of type T_i, add the sequence to *seqs*, and use a name from the sequence.

```
public class A {              public class B {
  public A() {...}              public B(int i) {...}
  public B m1(A a1) {...}       public void m2(B b, A a) {...}
}                             }
```

sequence s_1	sequence s_2	sequence s_3
`B b1 = new B(0);`	`B b2 = new B(0);`	`A a1 = new A();` `B b3 = a1.m1(a1);`

seqs	*names*	*extend*(`m2`, *seqs*, *names*)
$\langle s_1, s_3 \rangle$	`b1, b1, a1`	`B b1 = new B(0);` `A a1 = new A();` `B b3 = a1.m1(a1);` `b1.m2(b1,a1);`
$\langle s_3, s_1 \rangle$	`b1, b1, a1`	`A a1 = new A();` `B b3 = a1.m1(a1);` `B b1 = new B(0);` `b1.m2(b1,a1);`
$\langle s_1, s_2 \rangle$	`b1, b2, null`	`B b1 = new B(0);` `B b2 = new B(0);` `b1.m2(b2,null);`

Figure 7-4: Three example applications of the *extend* operator.

- If `null` is allowed, use `null`.

The new sequence is the result of concatenating the sequences in *seqs*, and appending a new call of method $m(T_1 \ldots T_k)$, using as input the names in *names*. At this point, RANDOOP has created a new sequence in the image of the component set. RANDOOP checks that the same sequence has not been created before. If it has, it discards the sequence and attempts to create a new sequence.

If the sequence has not been created before, RANDOOP executes each method call in the sequence, checks the API contracts, and checks for illegal behavior, after each call. If the sequence leads to a contract violation, it is output as a test case. If the sequence leads illegal behavior, it is discarded. Otherwise, RANDOOP applies the heuristic index pruning to the sequence. If one or more indices are still reusable after index pruning, RANDOOP adds the sequence to the component set. (Note that this process is slightly different from the procedure DRT$_S$ from Chapter 5, in that the determination of an illegal or equivalent sequence happens *after* the sequence has been generated.)

The latest version of RANDOOP continues appending random method calls to a newly-created sequence. This makes more efficient use of the objects that have already been created. If this flag is given, RANDOOP stops after the sequence reaches the length limit (which can also be specified by the user), or the sequence results in illegal or error-revealing behavior.

Repeating an Operation

Sometimes, a good test case needs to call a given method multiple times. For example, repeated calls to `add` may be necessary to reach code that increases the capacity of a container object, or repeated calls may be required to create two equivalent objects that can cause a method like `equals` to go down certain branches. To increase the chances that such cases are reached, we build repetition directly into the generator, as follows. When generating a new sequence, with probability N, instead of appending a single call of a chosen method m to create a new sequence, the generator appends M calls, where M is chosen uniformly at random between 0 and some upper limit max. (max and N are user-settable; the default values are $max = 100$ and $N = 0.1$.) There are other possible ways to encourage repetition in the generation process. For example, it could occasionally repeat input parameters, or entire subsequences.

7.6 Efficiency

This section describes aspects of RANDOOP's implementation that have a significant impact on the efficiency of the tool. Discussing these details of RANDOOP's implementation highlights engineering decisions important to obtaining an efficient implementation of directed random testing.

Reflection-based execution

Executing a new sequence consists in invoking the methods in the sequence with the appropriate input arguments. We considered two ways of doing this. The first was to output a sequence as source code, compile it, and execute it in a separate process. The second was to execute the sequence using Java (or .NET's) reflection mechanism [40, 74], which allows programmatic access to the methods and classes under test from within RANDOOP. The compile/run approach is safer, because it separates the execution of RANDOOP code and arbitrary code under test. Using reflection, on the other hand, means that both RANDOOP code and code under test are executed in the same process, and any undesirable behavior (e.g. infinite loops or stack overflows) resulting from executing calls of methods under test under random scenarios can affect the behavior of RANDOOP.

However, compiling and executing every method sequences separately is at least one order of magnitude slower than executing method sequences within the same process as RANDOOP. For this reason, RANDOOP uses reflection to execute method sequences. To have some degree of control over non-terminating executions in test code, which can freeze RANDOOP, the user can give a command-line argument that causes RANDOOP to execute every sequence in a different thread and kill threads that take longer than a default time limit. The use of threads slow the execution of RANDOOP, but not nearly as much as separate compilation and execution of test inputs.

Efficient sequence representation and concatenation

Method sequences are the basic data structure in RANDOOP, and sequence concatenation is a frequently-performed operation. If done inefficiently, it can significantly slow down the performance of the tool. The initial implementation of RANDOOP represented a sequence directly as a list of operations, and performed concatenation by creating a new list, copying the operations from the concatenated sequences. When profiling RANDOOP, we observed that concatenation took up a large portion of the tool's running time, and the component set quickly exhausted the memory available to the Java process.

To improve memory and time efficiency, we modified the internal representation of sequences. Instead of a list of operations, we implemented a sequence as a list of *sub-sequences*; the sequence constructor now took as input the sub-sequences making up the new sequence. To allow for safe sharing of sub-sequences, we made sequences immutable [15], and had a sequence refer to its constituent sequences without copying them. With these modifications, a new sequence uses space proportional to the number of constituent sub-sequences (instead of space proportional to the total number of operations), and concatenation takes time proportional to the number of constituent sub-sequences (instead of the total number of operations). The improvement reduced the memory footprint and time spent creating and concatenating sequences by more than an order of magnitude.

Efficient selection from the component set

When creating a new sequence for a method $m(T_1 \ldots T_k)$, RANDOOP searches the component set (up to k times) for sequences that create a values of type T_1 through T_k. The component set typically contains thousands of sequences; finding the subset of sequences that yield values of a given type can be time-consuming. To make the process faster, RANDOOP maintains the component set not as a single set of sequences, but as a *sequence map*, each entry mapping a specific type to the sets of all sequences that create one or more values of *exactly* the given type. The latter requirement is to prevent the sequence map from containing sets of sequences with many common sequences; for example, without the requirement, the entry for `Object` would contain every sequence. The map is updated appropriately when a new sequence is added to the component set.

A naïve way of collecting all sequences for a type T is to start with an empty set of sequences, S, and traverse every key in the sequence map, adding all the sequences at a given key to S whenever the key is type-compatible with T. However, repeated traversals of the key set are a large source of inefficiency (as our performance profiling showed).

To improve performance, RANDOOP maintains along with a sequence map a *subtype map* that represents the subtype relationship between all the types in the sequence map. More specifically, the subtype map gives, for every type T that is a key in the sequence map, a *set* of types consisting of all the subtypes (transitively) of T that are also keys in the sequence map. (Notice that the subtype map does not

contract class for reflexivity of equality

```
// Checks reflexivity of equals: "o != null => o.equals(o)==true"
public final class EqualsReflexive implements UnaryObjectContract {

   // Returns true if o is null or o.equals(o) returns true.
   // Returns false if o.equals(o)==false or if it throws an exception.
   public boolean check(Object o) {
     if (o == null) return true;
     try {
         return o.equals(o);
     } catch (Throwable e) {
         return false;
     }
   }
 }
```

Figure 7-5: Contract-checking class that checks reflexivity of equality for objects. The crucial method in this interface is `boolean check(Object)`, which must return `true` if the given object satisfies the contract.

represent a type tree; rather, it maps a type to the set of all its transitive subtypes, i.e. all the nodes reachable in its subtype tree.) Now, to find all the sequences that create values of a given type, Randoop first uses the subtype map to find all the relevant subtypes in the sequence map, and accesses those entries directly in the sequence map. The modification causes a substantial decrease in the time Randoop spends selecting sequences from the component set.

For historical reasons, Randoop for .NET does not perform this optimization.

7.7 Extensibility

Randoop provides a mechanism for a user to extend the tool in various ways. The most basic way of extending Randoop is by writing classes that implement Randoop's *sequence visitor* and *sequence decoration* interface, which are instances of the visitor and decorator design patterns [36]. A sequence visitor is invoked before and after every execution of a method in a new sequence that is being executed. The visitor can inspect the evolving execution, including the state of the objects created in the sequence. A sequence visitor can also record observations about a sequence and add them as decorations on a specific index of the sequence.

Much of Randoop's functionality is implemented using sequence visitors and decorations, including its contract-checking functionality. Randoop implements a contract-checking sequence visitor that checks the default object and method contracts (Figure 7-3). The contracts themselves are classes that implement special interfaces; for example, the interface `randoop.UnaryObjectContract` represents an object contract that checks a property of a single object. Figure 7-5 shows the class that implements the contract for reflexivity of equality. To add a new contract to Randoop, the user creates a new class that implements a contract-checking interface (e.g. `randoop.UnaryObjectContract`), and calls Randoop passing the name of the class in a command-line argument. On startup, Randoop creates a new contract-checking visitor and adds the default, plus any user-specified contracts, to the list of contracts to check. As a sequence executes, the contract-checking visitor checks the contracts, and if a contract violation occurs, the visitor adds a decoration to the sequence indicating the violation of the contract.

Two other features of Randoop that are implemented using sequence visitors are regression oracle creation and the creation creation of compilable test cases. When executing a sequence, Randoop uses a regression-creating visitor that records the return values of methods and adds decorations recording the values. When outputting a JUnit test case, Randoop uses a formatting visitor that outputs the sequence, along with any decorations, as compilable code.

A user wishing to create a new visitor to implement a specific task can create new classes implementing the sequence visitor (and sequence decorator) interfaces, and specify them in command-line arguments to Randoop, which will install the visitors and invoke them every time a new sequence executes. This allows a user to extend the tool in ways that go beyond the default visitors implemented.

Chapter 8

Experimental Evaluation

This chapter describes controlled experiments that compare the code coverage achieved and errors revealed by directed random testing, and compares the technique with other test generation techniques, including undirected random testing and exhaustive testing approaches based on model checking and symbolic execution.

Our experimental results indicate that directed random testing can outperform undirected random testing and exhaustive testing in terms of both coverage and error detection ability. Below is a summary of the experimental results described in the chapter.

- **Section 8.1.** On four container data structures used previously to evaluate five different exhaustive input generation techniques [92], inputs created with directed random testing achieves equal or higher block and predicate coverage [11] than all other techniques, including undirected random testing, model checking (with and without abstraction), and symbolic execution.

- **Section 8.2.** In terms of error detection, directed random testing reveals many errors across 14 widely-deployed, well-tested Java and .NET libraries totaling 780KLOC. Model checking using JPF [91] was not able to create any error-revealing test inputs: the state space for these libraries is enormous, and the model checker ran out of resources after exploring a tiny, localized portion of the state space. Our results suggest that for large libraries, the sparse, global sampling of the input space that directed random testing performs can reveal errors more efficiently than the dense, local sampling that a model checking approach performs.

- **Section 8.3.** To better understand the effectiveness of directed random testing compared with undirected random testing, we performed an experiment that compares the error-revealing effectiveness of directed random testing and two other random testing techniques, across hundreds of runs for each technique. Our results show that for some software artifacts, deep levels of the operation tree representing all sequences of operations have a higher density of error-revealing operations, making a technique that can generate longer sequences of operations more effective. Undirected random testing is often unable to generate long sequences due to illegal behaviors arising from executing operations with random inputs. Directed random testing's pruning heuristics allow it to generate sequences of operations that are both long and behaviorally diverse, resulting in greater error-revealing effectiveness.

- **Section 8.4.** To evaluate the post-processing techniques from Chapter 6, we applied replacement-based simplification and branch-directed generation to some of the subject programs from Section 8.2. Our results indicate that the simplification technique can reduce the size of a failing test case by an order of magnitude. The simple sequence-modification strategies that branch-directed generation employs led to a 65% success rate in flipping a frontier branch condition; however, the dynamic data flow analysis returned useful information for only a fraction of the frontier conditions, resulting in many uncovered frontier branches.

8.1 Coverage for Container Data Structures

Container classes have been used to evaluate many input generation techniques [4, 97, 96, 93, 92]. In [92], Visser et al. compared basic block and a form of predicate coverage [11] achieved by several input generation techniques on four container classes: a binary tree (`BinTree`, 154 LOC), a binomial heap (`BHeap`, 355 LOC), a fibonacci heap (`FibHeap`, 286 LOC), and a red-black tree (`TreeMap`, 580 LOC). They used a form of predicate coverage that measures the coverage of all combinations of a set of predicates manually derived from conditions in the source code. They compared the coverage achieved by six techniques: (1) model checking, (2) model checking with state matching, (3) model checking with abstract state matching, (4) symbolic execution, (5) symbolic execution with abstract state matching, and (6) undirected random generation. Coverage metrics, while not as compelling as error detection, may shed insight into the differences between techniques. For example, in code where no technique reveals an error, higher code coverage may nevertheless be an indication of greater effectiveness.

Visser et al. report that the technique that achieved highest coverage was model checking with abstract state matching, where the abstract state records the shape of the container and discards the data stored in the container. For brevity, we'll refer to this technique as *shape abstraction*. Shape abstraction dominated all other systematic techniques in the experiment: it achieved higher coverage, or achieved the same coverage in lesser time, than every other technique. Random generation was able to achieve the same predicate coverage as shape abstraction in less time, but this happened only for 2 (out of 520) "lucky" runs.

We compared directed random generation with shape abstraction. For each data structure, we performed the following steps.

1. We reproduced Visser et al.'s results for shape abstraction on our machine (Pentium 4, 3.6GHz, 4G memory, running Debian Linux). We used the optimal param-

		coverage				time (seconds)			
		JPF	DRT	JPF$_U$	RP$_U$	JPF	DRT	JPF$_U$	RP$_U$
block coverage	BinTree	.78	.78	.78	.78	0.14	0.21	0.14	0.13
	BHeap	.95	.95	.95	.86	4.3	0.59	6.2	6.6
	FibHeap	1	1	1	.98	23	0.63	1.1	27
	TreeMap	.72	.72	.72	.68	0.65	0.84	1.5	1.9
predicate coverage	BinTree	53.2	54	52.1	53.9	0.41	1.6	2.0	4.2
	BHeap	101	101	88.3	58.5	9.8	4.2	12	15
	FibHeap	93	96	86	90.3	95	6.0	16	67
	TreeMap	106	106	104	55	47	10	10	1.9

JPF	:	Best-performing of 5 systematic techniques in JPF.
DRT	:	Randoop: Directed random testing.
JPF$_U$	:	Undirected random testing implemented in JPF.
RP$_U$	:	Undirected random testing implemented in Randoop.

Figure 8-1: Basic block coverage (ratio) and predicate coverage (absolute) achieved by four input generation techniques.

eters reported in [92], i.e., the parameters for which the technique achieves highest coverage.

2. We ran Randoop on the containers, specifying the same methods under test as [92]. Random testing has no obvious stopping criterion; we ran Randoop for two minutes (its default time limit).

3. To compare against unguided random generation, we also reproduced Visser et al.'s results for random generation, using the same stopping criterion as [92]: generation stops after 1000 inputs.

4. To obtain a second data point for unguided random generation, we ran Randoop a second time, turning off all filters and heuristics.

As each tool ran, we tracked the coverage achieved by the test inputs generated so far. Every time a new unit of coverage (basic block or predicate) was achieved, we recorded the coverage and time. To record coverage, we reused Visser et al.'s experimental infrastructure, with small modifications to track time for each new coverage unit. For basic block coverage, we report the ratio of coverage achieved to maximum coverage possible. For predicate coverage, we report (like Visser et al. [92]) only absolute coverage, because the maximum predicate coverage is not known. We repeated each run ten times with different seeds, and report averages.

Figure 8-1 shows the results. For each technique/container pair, we report the maximum coverage achieved, and the time at which maximum coverage was reached, as tracked by the experimental framework. In other words, the time shown in Figure 8-1 represents the *time that the technique required in order to achieve its maximum coverage*—after that time, no more coverage was achieved in the run of the tool. The tool may have continued running until it reached its stopping criterion: on average, each run of JPF with shape abstraction took a total of 89 seconds; the longest run was 220 seconds, for `TreeMap`. Every run of RANDOOP took 120 seconds, its default time limit.

For `BinTree`, `BHeap` and `TreeMap`, directed random generation achieved the same block and predicate coverage as shape abstraction. For `FibHeap`, directed random generation achieved the same block coverage, but higher predicate coverage (96 predicates) than shape abstraction (93 predicates). Undirected random testing was competitive with the other techniques in achieving block coverage. For the more challenging predicate coverage, both implementations of undirected random testing always achieved less predicate coverage than directed random generation.

We should note that the container data structures are nontrivial. For `BHeap`, to achieve the highest observed block coverage, a sequence of length 14 is required [92]. This suggests that directed random generation is effective in generating complex test inputs—on these data structures, it is competitive with systematic techniques.

`FibHeap` and `BHeap` have a larger input space than `BinTree` and `TreeMap` (they declare more public methods). It is interesting that for `FibHeap` and `BHeap`, directed random generation achieved equal or greater predicate coverage than shape abstraction, and did so faster (2.3 times faster for `BHeap` and 15.8 times faster for `FibHeap`), despite the higher complexity. This suggests that directed random generation is competitive with systematic generation even when the state space is larger. (The observation holds for the much larger programs used in Section 8.2).

Interestingly, repetition of method calls (Section 7.5) was crucial. When we analyzed the inputs created by directed random generation, we saw that for `FibHeap` and `TreeMap`, sequences that consisted of several element additions in a row, followed by several removals, reached predicates that were not reached by sequences that purely randomly interleaved additions with removals. This is why undirected random generation achieved less coverage.

Rostra and Symstra

Two other systematic techniques that generate method sequences for containers are Rostra [96] and Symstra [97]. Rostra generates tests using bounded exhaustive generation with state matching. Symstra generates method sequences using symbolic execution and prunes the state space based on symbolic state comparison. Comparing RANDOOP with these tools would provide a further evaluation point. Unfortunately, the tools were not available to us. However, the data structures used to evaluate the tools were available. We report the results of running RANDOOP on these data structures, and provide a qualitative comparison with the results published in the Rostra and Symstra papers [96, 97].

Figure 8-2 shows the coverage achieved by the techniques, as reported in the published Rostra and Symstra papers. The figure also shows the result of running RANDOOP on the data structures. We arbitrarily chose 1 second (on a 3.2 GHz Pentium 4 CPU with 1 GB of memory) as the generation limit for RANDOOP. Since the experiments were performed at different times and different processors, we should not draw any hard conclusions from the run times. However, to get a feel of the rela-

			Rostra		Symstra		Randoop	
benchmark	LOC	branches	cov	time	cov	time	cov	time
IntStack	30	9	67%	689s	100%	0.6s	100%	1s
UBStack	59	13	100%	269s	100%	10s	100%	1s
BST	91	34	100%	23s	100%	317s	94%	1s
BinHeap	309	70	84.3%	51s	91.4%	8.9s	84%	1s
LinkedList	253	12	100%	412s	100%	0.95s	100%	1s
TreeMap	370	170	84.1%	42s	84.1%	63s	81%	1s
HeapArray	71	29	75.9%	3.7s	100%	11s	100%	1s

Figure 8-2: Coverage achieved by different test generation techniques. Rostra and Symstra results are as reported in [96, 97].

tive performance, we can normalize the run times based on the performance numbers of the different processors on the standard SPEC2000 integer benchmarks [51]. This would mean multiplying the run times of Rostra and Symstra columns (both of which were run on a 2.8 GHz Pentium 4 CPU with 512 MB of memory) by a factor of 0.8.

Based on Figure 8-2 and the approximate normalization above, we can claim that directed random testing achieves high coverage in a short time, at least for these data structures. It also appears to be quite competitive when compared to the Rostra and the Symstra tools.

The best measure to evaluate input generation techniques is error detection, not coverage. Our results suggest that further experimentation is required to better understand how systematic and random techniques compare in detecting errors in data structures. The next section evaluates directed random testing's error-detection ability on larger, widely-used component libraries, and compares it with exhaustive and undirected random generation.

8.2 Error-Revealing Effectiveness

Measuring coverage on small container data structures does not yield much insight into the relative strengths and weaknesses of different random and systematic techniques. To gain a better understanding of these, we performed a second experiment. The goal of the experiment was to determine the effectiveness of directed random testing in revealing previously-unknown errors in realistic-size software libraries, as well as compare its effectiveness with other techniques, including undirected random testing and systematic testing. To this end, we applied Randoop, the JPF [91] model checker, and JCrasher, an undirected random test generator, to 14 large, widely-used libraries (Figure 8-3), which together comprise a total of 780KLOC. The goal was to use these tools to reveal errors resulting from violations of basic object contracts, i.e. reflexivity of equality.

This section discusses the application of the different tools to find errors in the libraries, and reports the number and nature of the errors that each tool revealed.

Java libraries	**LOC**	**public classes**	**public methods**	description
Java JDK 1.5				
java.util	39K	204	1019	Collections, text, formatting, etc.
javax.xml	14K	68	437	XML processing.
Jakarta Commons				
chain	8K	59	226	API for process flows.
collections	61K	402	2412	Extensions to the JDK collections.
jelly	14K	99	724	XML scripting and processing.
logging	4K	9	140	Event-logging facility.
math	21K	111	910	Mathematics and statistics.
primitives	6K	294	1908	Type-safe collections of primitives.

.NET libraries	**LOC**	**public classes**	**public methods**	
ZedGraph	33K	125	3096	Creates plots and charts.
.NET Framework				
Mscorlib	185K	1439	17763	.NET Framework SDK class libraries. Provide access to system functionality and designed as foundation on which .NET applications, components, and controls are built.
System.Data	196K	648	11529	
System.Security	9K	128	1175	
System.Xml	150K	686	9914	
Web.Services	42K	304	2527	

Figure 8-3: Libraries used for evaluation.

Section 8.2.2 describes the application of Randoop, Section 8.2.4 describes the application of JPF, and Section 8.2.5 describes JCrasher [22], as well as Randoop *without* using pruning heuristics, as a second representative of random testing.

8.2.1 MinUnit

To reduce the amount of test cases we had to inspect, we implemented a test runner called MinUnit, which can replace JUnit or NUnit. JUnit and NUnit show all failing tests; MinUnit shows only a subset of all failing tests. MinUnit partitions the failing tests into equivalence classes, where two tests fall into the same class if their execution leads to a contract violation after the same method call. For example, two tests that exhibit a contract failure after a call to the JDK method `unmodifiableSet(Set)` belong to the same equivalence class. This step retains only one test per equivalence class, chosen at random; the remaining tests are discarded.

8.2.2 Directed Random Testing

For each library, we performed the following steps:

1. We ran Randoop on a library, specifying all the public classes as targets for testing. We used Randoop's default parameters (default contracts, heuristic pruning strategies, and 2 minute time limit). The output of this step was a test suite.

2. We compiled the test suite and ran it with MinUnit.

3. We manually inspected the failing test cases reported by MinUnit.

For each library, we report the following statistics.

1. **Test cases generated.** The size of the test suite (number of unit tests) output by Randoop.

2. **Violation-inducing test cases.** The number of violation-inducing test cases output by Randoop.

3. **MinUnit reported test cases.** The number of violation-inducing test cases reported by MinUnit (after reduction and minimization) when run on the Randoop-generated test suite.

4. **Error-revealing test cases.** The number of test cases reported by MinUnit that revealed an error in the library.

5. **Errors.** The number of distinct errors uncovered by the error-revealing test cases. We count two errors as distinct if fixing them would involve modifying different source code. The next section describes in greater detail the process we used to determine if a failure revealed an error.

6. **Errors per KLOC.** The number of distinct errors divided by the KLOC count for the library.

Determining Errors

Determining that a failure reveals an error can be difficult, due to the lack of complete specifications. In counting errors, we followed a conservative approach, erring on the side of *not* counting a failure as an error unless the failure explicitly violated documented behavior. We determined that a failure revealed an error as follows.

- *JDK libraries.* We labeled a test case as error-revealing only if it violated an explicitly stated property in the Javadoc documentation for the method in question. If the failure appeared to reveal problematic behavior, but there was no written documentation regarding the behavior in the Javadocs, we did not count the failure as an error.

- *Jakarta libraries.* The Jakarta Commons libraries are not as extensively documented as the JDK and .NET framework. We did not count as error-revealing test cases that appeared to us as indicative of problems in the library (or its design) but where the code in question lacked documentation.

library	**test cases generated**	**violation-inducing test cases**	**MinUnit reported test cases**	**error-revealing test cases**	**errors**	**errors per KLOC**
Java JDK						
java.util	22,474	298	20	19	6	.15
javax.xml	15,311	315	12	10	2	.14
Jakarta Commons						
chain	35,766	1226	20	0	0	0
collections	16,740	188	67	25	4	.07
jelly	18,846	1484	78	0	0	0
logging	764	0	0	0	0	0
math	3,049	27	9	4	2	.09
primitives	49,789	119	13	0	0	0
ZedGraph	8,175	15	13	4	4	.12
.NET Framework						
Mscorlib	5,685	51	19	19	19	.10
System.Data	8,026	177	92	92	92	.47
System.Security	3,793	135	25	25	25	2.7
System.Xml	12,144	19	15	15	15	.10
Web.Services	7,941	146	41	41	41	.98
Total	208,503	4200	424	254	210	

Figure 8-4: Statistics for test cases generated by Randoop. Section 8.2.2 explains the metrics.

- *.NET libraries.* The .NET libraries we tested had in general stronger specifications than that Java libraries: the documentation for the libraries explicitly specifies that every public methods should respect the contracts in Figure 7-3. We labeled each distinct method that violated a contract as an error for the .NET programs: a method that leads to the contract violation either contains an error, fails to do proper argument checking, or fails to prevent internal errors from escaping to the user of the library. Because MinUnit reports one test case per such method, MinUnit-reported test cases coincide with error-revealing test cases for .NET.

  We confirmed with the engineers responsible for testing the .NET libraries that any public method violating one of the contracts we checked would be considered an error.

Errors Revealed

Figure 8-4 shows the results. Randoop created a total of 4200 distinct violation-inducing test cases. Of those, MinUnit reported approximately 10% (and discarded the rest as potentially redundant). Out of the 424 tests that MinUnit reported, 254 were error-revealing. (This number is according to our determination of errors described above.) The other 170 were illegal uses of the libraries or cases where the contract violations were documented as normal operation. The 254 error-revealing test cases pointed to 210 distinct errors. Below are examples of the errors that Randoop revealed for different libraries.

- **JDK libraries.** Eight methods in the JDK create collections that return `false` on `s.equals(s)`. These eight methods shared some code, and together they revealed four distinct errors. The methods `XMLGregorianCalendar.toString()` and `XMLGregorianCalendar.hashCode()` crash because they do not handle cases where some fields hold legal corner-case values.

- **Jakarta commons libraries.** In `math` (2 errors), a matrix class has a field that holds the matrix data. One of the constructors leaves the field null, which is a legal, distinct state: a number of the methods implement special cases if the field is null. The check is omitted from `hashCode`. The second error in `math` is similar. In `collections` (1 error), an iterator initialized with zero elements (which is legal, according to the documentation) throws a `NullPointerException` when its `next()` method is invoked. The other 3 errors actually re-discover the errors that Randoop revealed for `java.util`, which was tested indirectly because the collections classes use functionality from that library.

  Most of the test cases that Randoop generated for the Commons libraries were not error-revealing. The main reason was that many classes in the libraries require specific sequences of method calls to operate properly, and random generation created many invalid sequences. For example, most of the iterators implemented in `collections` declare a parameterless constructor, but an iterator thus constructed must be followed by a call to `setCollection(Collection)` that specifies the collection to iterate over. Until this call is made, the iterator will not function, and it will throw an exception of type `NullPointerException` if `next()` or `hasNext()` is invoked. Randoop reported many contract violations due to incorrect use of these constructors.

- **.NET libraries.** In the .NET libraries, 155 errors are `NullReferenceExceptions` in the absence of `null` inputs, 21 are `IndexOutOfRangeExceptions`, and 20 are violations of `equals`, `hashCode` or `toString` contracts. For the .NET libraries, we used the .NET design guidelines for class library developers, which state that public library methods should never throw null reference or index-out-of-range exceptions. For System.Xml, we confirmed with the test team that this guideline is followed and that such exceptional behavior is tested, considered erroneous operation, and corrected if found. Given that null was never passed as an argument, such exceptions, if present, represent more serious errors.

  Randoop also led us to discover nonterminating behavior in System.Xml: when executing a sequence representing a legal use of the library, the last method call went into an infinite loop. When we reported this error, it was assigned the highest priority ranking (it can render unusable an application that uses the library in a legal manner) and was fixed almost immediately.

8.2.3 Executing Past Exceptions

As described in Chapter 7, Randoop does not generate method sequences that continue executing past (internally uncaught) exceptions. If it ever generates a sequence that results in an exception, Randoop discards the sequence and never extends it. The rationale for this decision is that exceptions typically indicate an abnormal condition occurring during execution, and executing past exceptions is less likely to reveal errors and more likely to result in arbitrary but non-error-revealing behavior.

A reasonable question is whether exploring method sequences that throw exceptions in the middle may actually be beneficial: could executing methods after an uncaught exception result in more errors, or different kinds of errors? One hypothetical possibility is that the errors that Randoop reveals are revealed more easily (i.e. faster) via "illegal" states that follow exceptional behavior, than via legal states. Another possibility is that a tool that executes method calls after an exception occurs reveals new *kinds* of errors: those resulting from objects being left in invalid states after an exception.

To shed light on these questions, we modified Randoop so that it continues extending inputs after an exception is thrown. Below, we call the modified version Randex. When a newly-generated test input leads to an exception, Randex adds it to the component set of inputs that can be extended to yield longer inputs. (The original Randoop discards exception-throwing inputs and never extends them.) This modification causes Randex to generate, execute, and output test cases that throw exceptions in the middle but continue executing, exemplified by the error-revealing test case shown below in Figure 8-5, output during a run of Randex on the Java JDK.

We reproduced the experiment from Section 8.2.2, using Randoop's default oracles and generation time limit (2 minutes), and comparing with running Randex using the same parameters.[1] The results of this experiment were as follows.

- **Randex is effective in revealing errors.** Randex is similar to Randoop in its end-to-end behavior. Randex output 224 tests which revealed 20 distinct errors, while Randoop output 233 tests which revealed 23 errors. Both tools had similar "error yields," or ratios of failing tests reported to distinct errors revealed by those tests. For some programs, Randex had a higher number of false positive tests, and redundant tests that revealed the same underlying error. But this did not hold for all libraries: for other libraries, it was the opposite. In summary, there was no discernible pattern in terms of the number of redundant tests or false positives generated by Randoop vs. Randex.

  This indicates that generating tests that execute methods after exceptions are thrown does not result in a large decrease in error-revealing effectiveness, at least when performing "short" runs of the tool. (Below we discuss potential

[1]This experiment was performed on a newer version of Randoop from that in Section 8.2.2, and the number of errors revealed were slightly higher than those shown in Figure 8-4 due to efficiency improvements in the new version. We also added two additional libraries from those in the original experiment.

```
public void Randex_test1() throws Throwable {

  Vector var1 = new Vector(1);
  IdentityHashMap var2 = new IdentityHashMap(10);
  Collection var3 = var2.values();
  var1.addElement(var3);
  Collections.reverse(var1);
  Vector var4 = new Vector(1);
  IdentityHashMap var5 = new IdentityHashMap(10);
  Collection var6 = var5.values();
  var4.addElement(var6);
  Iterator var7 = var4.iterator();
  boolean var8 = var1.addAll(var4);
  try {
    // THROWS ArrayIndexOutOfBoundsException
    Object var9 = var4.remove(-1);
  } catch (Throwable t) {
    // continue executing.
  }
  TreeSet var10 = new TreeSet(var4);
  Comparator var11 = var10.comparator();
  SortedSet var12 = Collections.unmodifiableSortedSet(var10);

  // RETURNS FALSE reflexivity of equals (var12)
  assertTrue(var12.equals(var12));
}
```

Figure 8-5: A test case generated by Randex. Unlike tests cases generated by Randoop, test cases generated by Randex can throw an exception, catch it, and continue execution after the exception.

decrease in effectiveness for longer runs.) This result is somewhat surprising: before running the experiment, we expected Randex to output a lot more failing tests cases that revealed no errors.

- **Randex has longer time to failure.** Randex does decrease the effectiveness of the generation in one respect: it increases the time to failure for any given error. The average time required by Randex to reveal a specific error is on average 70% greater than using Randoop; for some errors, Randex takes up to 10 times longer to reveal the same error as Randoop (the opposite is not the case).

  Why the longer time to failure? The increased time to failure is *not* due to a slowdown caused by exception-throwing method calls: both Randoop and Randex generate and execute roughly the same number of method calls and tests per unit of time, with Randex being only 8% slower. Randex *takes longer to find errors because it generates more redundant operations that modify no object state, wasting generation time.* We determined this by manually inspecting the tests output by Randex. Our analysis revealed that (at least for the 10 subject libraries) exception-throwing operations for the most part do not modify the state of their arguments. An example representative of the general case is the test case from Figure 8-5, which throws an `ArrayIndexOut-OfBoundsException`. The exception-throwing call does not mutate the state of its receiver (`var4`). In other words, RANDEX spends more time performing "no-op" operations that consume time but have no effect on the result, and thus do not contribute to the testing effort.

- **Randex does not reveal new or different kinds of errors.** The errors revealed when executing past exceptions were essentially a subset of the errors revealed when not executing past exceptions. This result is also partially explained by the fact that exception-throwing methods in these libraries most often did not mutate any object state. Thus, RANDEX did not reveal new errors resulting from an object being left in a modified and invalid state after an exception occurs, which is one class of errors we thought an approach like RANDEX might reveal and RANDOOP would not.

  However, this result holds for the libraries we used in the experiment, and the relatively small set of validity properties that RANDOOP checks (e.g. reflexivity of equals). RANDEX could very well reveal bugs resulting from invalid post-exception states, if we inspected the behavior of *all* tests that RANDEX generated (many thousands of them), not only those resulting in a violation of one of the small set of properties built into the tool. Or, for a different set of libraries (for which exceptions are more likely to leave objects in invalid states), RANDEX could be more effective in revealing these kinds of errors.

Execution Performance. During the experiment, we also collected other statistics of RANDOOP and RANDEX. The first statistic was the total number of operations performed (i.e. calls of public methods under test) during the run of each tool. In Java (and .NET), a method invocation that results in an exception can be slower than an invocation that completes normally, so we wanted to determine if generating test inputs that threw exceptions in the middle slowed down the tool considerably. As it turns out, it did not. RANDEX executed on average only 8% fewer operations than RANDOOP in the same period of time; both tools were able to generate and execute comparable number of test inputs.

Time to Failure. As we mentioned above, the main difference between RANDOOP and RANDEX was in the average time to failure. The time to failure for a particular error is the time it takes the tool to output a failing test case that reveals the error. To obtain time to failure data, we recorded the time at which each test case was output, and them computed the time to failure for different errors. On average, the time to failure for a particular error was 70% higher (took longer time) for RANDEX, compared with RANDOOP. The result appears to be independent of the random seed used, based on repeating runs of RANDOOP and RANDEX using different seeds.

Executing past exceptions increases the time it takes the tool to reveal the error. This is not necessarily a problem for errors that are revealed within minutes—the user can just run the tool a few minutes longer. It may make the difference for very rare errors that require hours, or days, to be revealed; in that case, taking twice as long may make the difference between discovering or missing the error.

8.2.4 Systematic Testing

To compare directed random testing with systematic testing, we used the Java PathFinder model checker [91] (JPF) to test the Java libraries. JPF is an explicit-state software model checker for Java that executes code in a specialized virtual machine, and systematically explores all potential execution paths of a Java program to find violations of properties like deadlocks, unhandled exceptions, etc.

JPF does not actually create method sequences. To make it explore method sequences, the user has to manually write a "driver" program that nondeterministically calls methods of the classes under test, and JPF explores method sequences by exploring the driver (for instance, Visser et al. wrote driver programs for the container experiments [92]). We wrote a *driver generator* which, given a set of classes, creates a driver that explores all possible method sequences up to some sequence length, using only public methods. For this experiment, we augmented the drivers with the code that checked the same contracts as Randoop (Figure 7-3). We performed the experiments on a Pentium 4, 3.6GHz, 4G memory, running Debian Linux.

For each library, we generated a universal driver and had JPF explore the driver until it ran out of memory. We specified sequence length 10 (this was greater than the length required to find all the Java errors from Figure 8-4). We used JPF's breadth-first search strategy, as done for all systematic techniques in [92]. In that paper, Visser et al. suggest that breadth-first search is preferable than depth-first search for this kind of exploration scenario. We used JPF's default state matching (shape abstraction is not currently implemented in JPF, other than for the four containers from Section 8.1).

Figure 8-6 shows the results. Using breadth-first search, JPF ran out of memory on all libraries without reporting any errors. Considering the size of the libraries, this is not surprising, as JPF was barely able to explore the libraries before state space explosion became a problem.

jCUTE [86] performs *concolic testing*, a systematic technique that performs symbolic execution but uses randomly-generated test inputs to initialize the search and to allow the tool to make progress when symbolic execution fails due to limitations of the symbolic approach (e.g. native calls). Comparing directed random generation with concolic testing would be interesting. Unfortunately, jCUTE crashed when compiling the drivers generated for the classes because it could not handle drivers of the size generated for our subject programs.

8.2.5 Undirected Random Testing

To measure the benefits of directed random testing versus undirected random testing, we reran Randoop as described in Section 8.2.2 a second time, using the same parameters, but disabling pruning. Across all libraries, unguided generation created 1,326 violation-inducing test cases. Out of these, MinUnit reported 60 test cases,

	heuristic breadth-first search			depth-first search		
library	time and termination cause	violation inducing test cases	distinct errors	time and termination cause	violation inducing test cases	distinct errors
java.util	33s (M)	0	0	12,000s (N)	0	0
javax.xml	22s (M)	0	0	1016s (E)	24	0
chain	29s (M)	0	0	72s (M)	0	0
collections	50s (M)	0	0	1165s (E)	78	0
jelly	28s (M)	0	0	4s (E)	1	1
logging	57s (M)	0	0	22s (E)	0	0
math	2s (E)	0	0	1s (E)	0	0
primitives	33s (M)	0	0	12,000s (N)	0	0

Termination causes

N:	JPF stopped exploration after reaching the time limit (12,000 seconds) .
M:	JPF ran out of memory before reaching the time limit.
E:	JPF crashed before reaching the time limit.

Figure 8-6: Results of running JPF on the Java libraries, using two search configurations.

all of which pointed to distinct errors (58 in the .NET libraries, and 2 in the Java libraries). Below are examples of the errors that Randoop was able or unable to reveal when running without pruning.

- Undirected generation did not find any errors in `java.util` or `javax.xml`: it did not create an input complex enough to expose the violation of `equals` by eight methods in `Collections`, and did not create a test input that caused `hashCode` or `toString` to throw an exception for `XMLGregorianCalendar`.

- Unguided generation was able to generate the test cases that reveal two errors in `math`. This is not surprising, because the errors manifest themselves immediately after a call to a public builder. On the other hand, it was not able to discover the `java.util` errors that directed generation discovered when testing the `math` library.

- In the .NET Framework libraries, the number of methods that were shown to throw forbidden exceptions went down from 192 (with directed generation) to 58 with unguided generation, and Randoop and was unable to create the sequence that uncovered the infinite loop in `System.Xml` (to confirm that this was not due simply to an unlucky random seed, we ran Randoop multiple times using different seeds; undirected generation never found the bug).

We also measured how many distinct objects Randoop created during its 2-minute run, as measured by the `equals` method (i.e., objects for which `equals` returns false). With directed generation, Randoop created an average of 1271 distinct objects per tested library. Without directed generation, Randoop created an average of 268 distinct objects.

`IllegalArgumentException`	332
`NullPointerException`	166
`NumberFormatException`	2
`NegativeArraySizeException`	3
`ClassCastException`	6
`MissingResourceException`	8
`ArrayIndexOutOfBoundsException`	77
`RuntimeException`	1
`IllegalAccessError`	1
`IndexOutOfBoundsException`	2

Figure 8-7: Exception types that JCrasher reported. The numbers reported is the number of test cases that JCrasher reported as potentially error-revealing due to an exception of the given type. All exceptions are from package `java.lang` except for `MissingResourceException`, which is in `java.util`.

As a second, independent comparison with undirected random testing, we also used JCrasher [22] to the libraries. JCrasher implements undirected random testing, with the goal of revealing exceptional behavior that points to errors. JCrasher randomly generates tests, then removes tests that throw exceptions not considered by JCrasher to be potentially fault-revealing. We used JCrasher to generate test cases for the Java libraries. JCrasher takes as input a list of classes to test and a "depth" parameter that limits the number of method calls it chains together. We ran JCrasher with maximum possible depth.

JCrasher ran for 639 seconds, created a total of 598 failing test cases. We used the same methodology to label error-revealing JCrasher test cases as for our tools. Out of the 598 failing test cases generated by JCrasher, 3 were error-revealing, and they revealed one distinct error. The error is one that RANDOOP did not find, not because of a limitation in its grammar of tests (the test case is easily expressible as a sequence) but because the random generation process did not happen to construct the specific input. JCrasher generated 595 non-error-revealing test cases. Figure 8 shows the types of exceptions that it reported. About half (332) were test cases that threw an `IllegalArgumentException` when given an illegal argument. 166 test cases threw a `NullPointerException` but the exception was caused because the value null was explicitly given as an input to a method. Figure 8-7 lists the exception types that JCrasher reported.

The rest of the test cases were illegal uses of the libraries. For example, 2 `NumberFormatException`s were thrown because JCrasher attempted to parse a number from a string that does not represent a number; 3 `NegativeArraySizeException`s were thrown because JCrasher passed a negative argument that is used to set an array's size; 6 `ClassCastException`s were thrown because the wrong type of argument was passed; 2 `ArrayIndexOutOfBoundsException`s were thrown because a negative index was passed as a parameter that indexed into an array; etc. (Note that in the .NET libraries, the specification says that a public method that throws an exception of type `IndexOutOfRangeException` is erroneous operation. The .NET guidelines require

IllegalArgumentException instead. This is not the case for the Java libraries, and Randoop does not report out-of-bounds exceptions as violations.) The test cases reported by JCrasher that threw out-of-bounds exceptions were cases of illegal parameters being passed. Our results suggest that JCrasher is not very effective in revealing contract violations like the ones that Randoop reveals.

8.3 Operation Depth and Effectiveness

The experimental results from the last two sections suggest that directed random testing can be more effective than other test input generation techniques in revealing errors and achieving code coverage. This section describes an experiment that explains a reason for the technique's improved effectiveness over undirected random testing.

When inspecting Randoop failures, we observed that many errors were revealed by relatively long sequence of operations (i.e. dozens of method calls), even though the errors, upon inspection, could be revealed by much smaller sequences (i.e. five or less calls). Another way of describing this observation is by thinking of an "operation tree" representing the space of operation sequences for a given software artifact, with the root of the tree representing the empty sequence, and each path of length N through the tree representing a sequence of N operations. (All the techniques we have discussed so far can be seen as exploring some portion of this tree.) A hypothesis that would explain the "long failing test" observation is that for some software artifacts, deeper levels of the operation tree have a higher proportion of error-revealing operations (nodes) than shallow levels. We are not the first to propose or evaluate this hypothesis: in a recent empirical study, Andrews et al. observed a similar phenomenon when applying random testing to flight software systems [7].

Figure 8-8 depicts the hypothesis graphically. It shows the first four levels of an operation tree. The tree shows three kinds of nodes, corresponding to normal operations, illegal operations, and error-revealing operations. We define the *failure rate at a given depth* as the ratio of error-revealing to total operations for a given depth. The example shows an example failure rate that increases for greater operation depths.

To shed light on the relationship between operation depth, failure rate, and error-revealing effectiveness of random testing techniques, and its implication for directed random testing, we gathered and analyzed failure data for three different techniques: undirected random testing, implemented as a random walk along the method sequence tree; bottom-up random generation with pruning based *only* on illegal inputs (i.e. procedure Bu/L from Chapter 4); and directed random testing. Because the results of a random testing technique can vary depending on the choice of random seed, we performed a large-scale experiment consisting of approximately 2,500 runs for each technique (totaling more than 1,000 CPU hours), using different seeds and subject programs.

In summary, the results of the experiment are as follows.

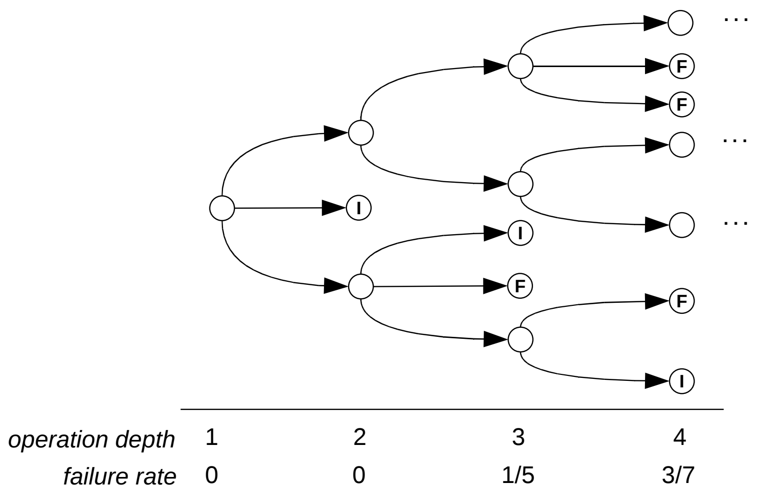

Figure 8-8: A tree depicting the space of operation sequences, and a *hypothetical* failure rate. The root of the tree represents the empty sequence; nodes at depth 1 represent sequences of length 1, etc. An "F" node indicates an error-revealing operation, and an "I" node presents an illegal operation. The failure rate at a given depth is the number of error-revealing operations divided by the total number of operations.

- The results substantiate the hypothesis that for some software artifacts, deeper levels of the operation tree have a higher failure rate than shallow levels.
- Undirected random testing techniques execute fewer operations in deeper areas of the tree (where the failure rate is higher). This is due to the decreased probability of executing a long series of *legal* continuous operations when using random inputs.
- Directed random testing is better able to execute operations in deeper portions of the tree because it concatenates legal sequences of operations, and achieves a higher failure rate due to equivalence pruning. This enables the technique to reveal more errors.

The rest of the section describes the experiment and results in greater detail.

8.3.1 Experimental Setup

To better understand the relationship between operation depth and effectiveness, as well as the effectiveness of directed random testing's different pruning strategies, we evaluated three techniques.

1. **Rw (Random Walk).** The random walk-based sequence generator (we will call the technique Rw from here on) works as follows. The technique starts

with an empty sequence s. It repeatedly extends the sequence with a new operation (chosen at random from all possible operations at that point), and executes the operation. If the sequence results in an error-revealing state, Rw outputs the sequence as a test case. If the sequences results in a (heuristically-determined, as in Randoop) illegal state, Rw discards the sequence. In both cases, Rw resets the sequence to the empty sequence, and begins a new series of random extensions. If an operation results in normal behavior, a new extension is performed.

An attractive property of Rw is that it has no tunable parameters, decreasing the risk that any results are due to specific parameter settings.

2. **Bu/L.** Bottom-up generation with illegal-input pruning (Section 4.1) is based on bottom-up generation but discards only illegal sequences; in particular, it does no pruning based on equivalent values. Comparing the results obtained by Bu/L and Drt allowed us to understand the contribution of pruning based on input legality.

3. **Drt.** The final technique we considered was the full version of Drt (Chapter 5), which uses a component set, discards illegal sequences, and uses index pruning heuristics based on value equality.

A difficulty in performing a large-scale experiment is the potential amount of human time require to inspect the resulting data. Inspecting a failing test case to determine if an error (and which error) is revealed can be a very time-consuming process; it can be even more difficult for mechanically-generated test cases. The thousands of runs of the techniques generated thousands of failing test cases, making manual inspection infeasible. Thus, we designed the experiment in a way that would allow for an automated, yet reasonably precise way of classifying failing test cases. The following sub-sections give details on how we did this.

Test Oracles

We limited the oracles to those that check for universal object properties that every object should respect, e.g. reflexivity and symmetry of `equals`, `equals` returning `false` on `null`, and `equals/hashCode` consistency, but excluded properties related to exceptional behavior, which leads to non-error-revealing failures more often than the above properties. Furthermore, we further strengthened the properties to fail only if incorrect but *non-exceptional* behavior was exhibited. For example, we counted a call of `o.equals(o)` returning `false` as a failure, but not if the call resulted in an exception. Based on our experience inspecting failing test cases in previous experiments, a violation of one of the above properties not involving an exception but rather returning the incorrect value almost always points to an actual error. Restricting the oracles in this way gave us greater confidence that reported failures were likely due to errors.

Subject Programs

As subject programs, we used five Java libraries. Three of them were `java.util`, `collections` and `primitives`, i.e. the libraries from Section 8.2's benchmark (Figure 8-3) that we knew contained at least one error fitting the description of our strengthened oracle. (We discarded the other libraries from the experiment after multiple runs of all three techniques revealed no errors.) The other two libraries were `trove` (278 classes, 75KLOC) and `jace` (164 classes, 51KLOC). These libraries declare many classes with `equals` and `hashCode` methods.

Stopping Criterion

We used each technique to generate test inputs for each of the five libraries. We gave each technique a limit of 100 million operations (distributed across multiple runs) for each library. We used operations instead of time to remove any unfairness due to implementation details of each technique; this was mostly to be fair to Rw (the random walk-based technique), which we spent longer time per operation, in small part due to implementation details, but mainly due to the fact that it led to more exceptional behavior than the other techniques, which resulted in a 1-3 order of magnitude slowdown, depending on the subject library. While this points out an efficiency advantage of directed random testing in the context of Java, we did not deem it to be the fundamental reason for directed random testing's effectiveness; so we normalized all techniques using operation counts to limit generation.

Parallel Runs

To collect the failure data, we wrote a script that for each technique/library pair, invoked multiple processes across 10 machines, each consisting of a run of Randoop specifying (1) the given technique, (2) the subject program, (3) a unique random seed, and (4) a stopping criterion for the individual run of Randoop. While in principle, Randoop could generate tests spanning 100 million operations in a single run, in practice, all three techniques tended to slow down after a while, as the garbage collector workload increased due to a large number of randomly-generated objects; recall that Randoop generates and executes test inputs in the same process. In the case of Bu/L and Drt the growing size of the component set also slowed down the tool after several minutes of generation. Thus, the script used a simple coverage-based stopping criterion that killed a given run after 100 seconds of no new code coverage. All three techniques typically reached the stopping point after a 10- to 30-minute run. Upon reaching stopping point, Randoop output all failing tests and other generation statistics. The script stopped launching new processes for a given technique when the 100 million operation limit was reached, which spanned 300-500 total runs for each technique/library pair.

	failing test cases			errors revealed		
library	Rw	Bu/L	Drt	Rw	Bu/L	Drt
`java.util`	2,700	4,400	19,000	20	21	27
`collections`	11,000	8,300	13,000	28	37	48
`primitives`	2,300	860	1,900	16	13	19
`trove`	110,000	140,000	330,000	20	27	27
`jace`	6,400	6,400	7,500	15	15	26
TOTAL	140,000	160,000	380,000	99	113	147

Figure 8-9: Failing test cases generated and errors revealed over all runs of the techniques. Due to the amount of test cases produced, we did not inspect them manually; we counted all failures resulting from the failure of a given method as revealing a single error (an inexact approximation to actual errors, but one we applied uniformly across all techniques) .

8.3.2 Results

Figure 8-9 shows the overall results of the experiment, including the total number of failing test cases and errors found by each technique. To count errors, we mechanically determined the `equals` method involved in every failure (i.e. the method that violates a correctness property), and counted all failures for a given method as revealing a single error. This is only an approximation to real errors; for example, a single method may contain many errors, and a method invocation may result in a failure due to an error in a previous invocation that corrupted an object's state. An informal inspection of a few dozen randomly-selected test cases suggests that this approximation is reasonable: in the test cases we have inspected, most of the time the error is located in the method that led to the violation. We used the same criterion across all techniques.

As the results show, Drt revealed the most errors, followed by Bu/L; Rw revealed the fewest errors; nevertheless, Rw revealed several errors.

Operation Distribution

To understand the relationship between operation depth and failures for the three techniques, we start by considering the total number of operations each technique performed at a given depth. The top three plots in Figure 8-10 show this information for each technique. As the figure shows, the number of operations Rw executes at a given depth decreases dramatically for higher depths. Calling an operation with random inputs arguments (at least for most object-oriented libraries) has a relatively high probability of resulting in an exception, and this makes the chances of Rw generating a long sequence of normally-executing operations decrease as sequence length increases. To see why this happens, imagine a simple model in which executing an operation with random inputs results in illegal behavior with probability p. Then, the probability of generating a legal sequence of n operations is $(1 - p)^n$. Even if the probability of illegal behavior is small, the probability of generating a long

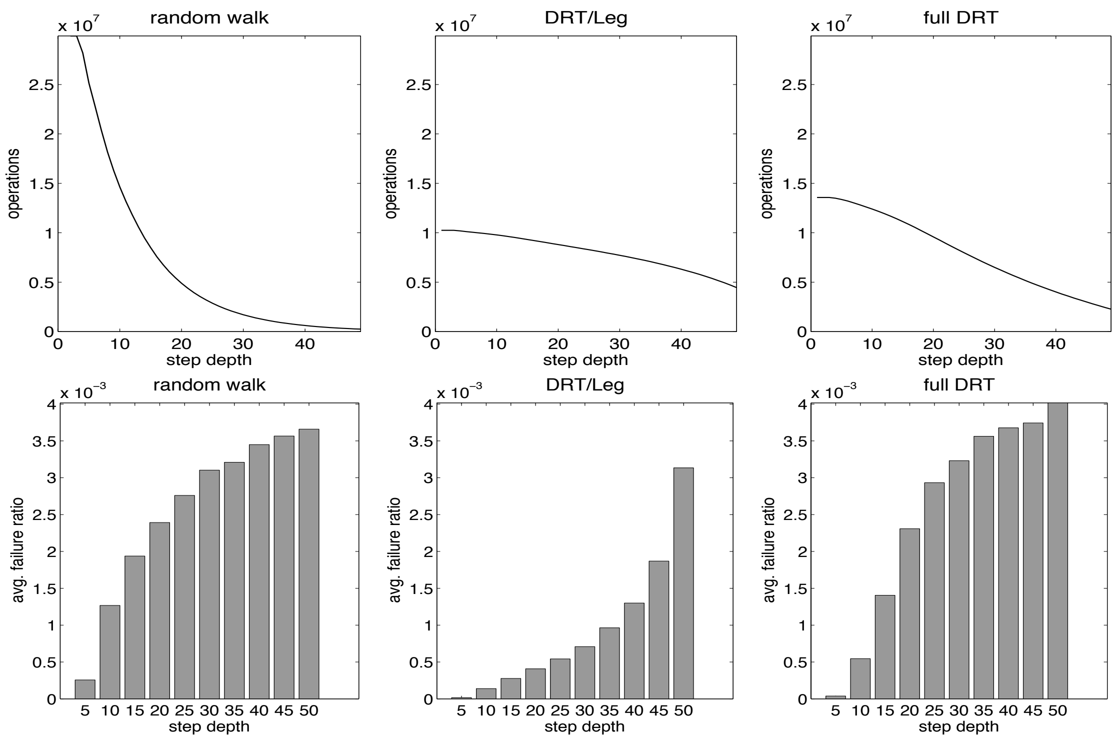

Figure 8-10: Top: distribution of operations performed at different depths for each technique. Bottom: average failure rate (averaged over all failures) for each technique.

sequence quickly decreases; e.g. for $p = 0.1$ and $n = 10$ the probably of generating an illegal sequence is 35%. For most libraries we have encountered, the probability of an exception given random input is much larger than 0.1. Thus, RW performs a large number of short random walks, and very few long ones.

BU/L and DRT also show a decrease in number of operations at higher depths, but the effect is less pronounced than for RW. This makes sense, since these techniques use a component set of legal sequence to build longer sequences, and by this process are able to generate longer sequences of legal operations.

These results by themselves do not explain the increased effectiveness of BU/L and DRT over RW. However, when combined with the failure data, it becomes clear why a random walk is less effective. Next, we discuss the failure rate data for each technique.

Failure Rate

We compute each technique's failure rate as an average of failure ratios for each individual error. Given a particular error, we computed the failure rate by dividing the number of operations that led to the error at the given depth by the total operations that the technique executed at that depth. The bottom plots in Figure 8-10 shows the average failure rates for all techniques.

For all techniques, the failure rate increases as operation depth increases, supporting the original hypothesis that deeper levels of the operation tree have a higher error ratio. While it supports the hypothesis, the data does not explain it. A complete explanation of this hypothesis is beyond the scope of the thesis; however, our manual inspection of many failing test cases suggests that many errors are triggered by a specific series of events, e.g. calling a specific operation with a given argument, and later on calling another specific method with the result of the previous operation; more sporadic errors require a longer (or more specific) series of events. Longer sequences of operations may increase the chances that the specific series of events triggering an error will happen, even if the series of events is itself short. A simple analogy is to think of a sequence as a sequence of dice rolls, and an error is triggered when rolling a series of two specific numbers, say a 6 followed by a 3, with other numbers allowed in between. In this scenario, rolling one long sequence of 100 rolls can be shown to have higher probability in revealing the error than rolling many short sequences.

Operation Distribution, Failure Rate, and Effectiveness

The increased error-revealing effectiveness of the replacement-based techniques over a random walk can be explained by considering the *combined* effects of each technique's operation distribution and failure rate. Rw executes more than half of its operations at depth less than 10, where its failure rate is smallest, while it executes very few sequences at depths in the 40-50 range, which is where its failure rate is highest. Rw generates mostly short sequences which have a smaller chance of revealing an error, and is rarely able to generate longer sequences that more likely to reveal errors.

Bu/L performs better than Rw, which is surprising if we only look at the technique's failure rate: Bu/L's average failure rate is lower at all depths than Rw's average failure rate. It is not surprising that Bu/L's failure rate is lower than that of Rw. Bu/L performs many repeated non-error-revealing operations when concatenating component sequences to yield a new sequence, while every operation that Rw performs is always chosen at random However, Bu/L is able to execute operations at deeper levels, where the chances of revealing an error are increased. Taken in combination, these factors give a small edge to Bu/L over Rw in error-revealing effectiveness.

Drt differs form Bu/L in that it performs heuristic pruning based on equivalent values. As the results show, this type of pruning increases the effectiveness of the technique, allowing it to overcome the decrease in effectiveness resulting from concatenating sequences with repetitive operations. At the same time, the technique is able to execute more operations at deeper levels, where the error ratio is higher. In other words, Drt combines the two other technique's advantages: a higher failure rate, and ability to execute deep operations.

Easy vs. Hard Errors

The errors revealed by each technique in the course of the experiment exhibited different numbers of revealing test cases generated, ranging from errors revealed by a single failing test case to errors revealed by more than 20,000 test cases. Informally, call an error *hard* for a particular random testing technique if it was revealed by a

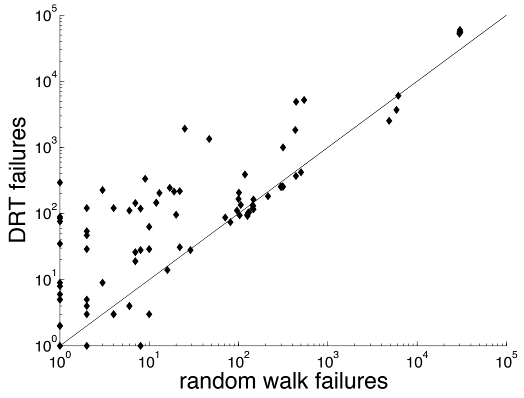

Figure 8-11: Number of DRT and random walk failures for each distinct error.

small number of test cases during the experiment. For example, an error revealed only once in 500 runs is hard for a technique, and unlikely to be revealed by in a single run.

Figure 8-11 shows a plot that compares the number of failures per error for RW and DRT. Each dot in the plot represents an error, the dot's x-coordinate gives the number of error-revealing test cases generated by RW, and the y-coordinate gives the number of error-revealing test cases generated by DRT. A dot along the diagonal is an error that is equally hard for both techniques. A dot above the diagonal is an error that is harder for RW than for DRT. Finally, a dot below the diagonal is an error that is harder for DRT than RW. Note that both coordinates are log-based.

As the plot shows, DRT dominates RW, in the sense that most errors were equally hard, or harder, for RW than for DRT. The effect is more pronounced for harder errors: the dots closer to the origin show a higher vertical spread than those farther from the origin. For example, dots along the y axis represent errors for with RW was only able to generate one error-revealing test case; for these errors, DRT generated from 1 to 300 test cases. In summary, the data indicates that, in general, errors that are hard for RW to reveal are easier for DRT to reveal, but not vice-versa.

8.4 Evaluation of Post-Processing Techniques

Replacement-Based Simplification

To evaluate replacement-based simplification, we applied the simplifier to all of the failing test cases from Section 8.2. The average size of a non-minimized test case was 42 method calls, and the average size of a minimized test case was 5 method calls. (These results are for the Java libraries; the minimization step is not implemented for .NET.) For example, the original length of the contract-violating test case in Figure 7-2 was 25 method calls, and its final length after simplification was 5 calls.

A more thorough evaluation of the technique is part of future work. In particular, it would be interesting to compare the results achieved by replacement-based simplification and those achieved by delta debugging or dynamic slicing, in terms of the time required by each technique to simplify a test case, and the resulting size of the test case. In our experience, removal-based simplification goes a long way towards minimizing the size of a test case, but replacement based simplification (and in particular, replacing values of complex types with simpler types, as the Java implementation aims to do) leads to test cases that make the cause of a failure more directly apparent.

Branch-Directed Generation

To evaluate branch-directed generation, we performed a case study in which we applied the technique to the output of RANDOOP on the Java collections, a subset of the Java JDK comprising 25 classes that includes collections classes including linked lists, stacks, hash tables, etc. as well as utility methods to sort or shuffle a collection, convert an array to a collection, etc. We implemented a coverage instrumentation tool that modifies source code to track branch coverage information. The tool reported a total of 1032 branch conditions across the 25 classes.

We ran RANDOOP on the coverage-instrumented versions of the classes, and stopped generation after 100 seconds passed with no new branches covered. At the end of the generation process, RANDOOP output all the sequences it generated, which we used as input to the branch-directed generator. The RANDOOP-generated input set led to 467 fully-covered branch conditions, 163 frontier conditions, and 402 fully-uncovered conditions. RANDOOP generated several thousand witness sequences for each frontier condition. Our current implementation of the data flow analysis is slow: it takes on the order of several seconds to minutes to instrument and analyze each sequence. To speed the process, we limited the analysis to only the 10 smallest sequences of each frontier condition.

Of the 163 frontier conditions, the data flow analysis returned non-empty influencing variable sets (for one or more sequences) for only 29 frontier branches. The analysis can fail to report influencing variables of an input for several reasons. The first reason is if there are no external variable in the sequence that can affect the result of the frontier branch condition. For example, consider the `HashMap()` constructor. It internally calls another constructor, `HashMap(int,float)`, that takes initial capacity and load factor parameters. `HashMap()` passes default values for these parameters. The boxed condition below in constructor `HashMap(int,float)` was a frontier branch condition (with the *false* branch covered):

```
 public HashMap(int initialCapacity, float loadFactor) {
  if (initialCapacity < 0)
    throw new IllegalArgumentException("Illegal capacity: + initialCapacity);
  if (initialCapacity > MAXIMUM_CAPACITY)
    initialCapacity = MAXIMUM_CAPACITY;
  ...
}
```

The sequence `m=new HashMap()` reaches the condition, but because it uses the 0-ary constructor, there is no possible modification strategy in the generator that can flip the result of the condition. On the other hand, the sequence `i=1; j=1;`

`m=new HashMap(i,j)` declares variables that can be successfully modified to flip the condition. A few of the instances in which the analysis returned no influencing variable information was because none of the 10 sequences used in the analysis declared relevant variables that influenced the result of the condition, like sequence `m=new HashMap()` in the example above.

Many other cases of missing information were due to the fact that the analysis only computes data flow information, not control-flow information, and some frontier conditions depended on the latter, not the former. This is not a fundamental limitation of branch-directed generation, but a limitation of our current dynamic analysis that could be addressed by tracking control flow as well as data flow information.

Of the 29 frontier conditions for which the analysis reported influencing variable information, the modification strategies resulted in a successful modification (i.e. a modification that resulted in the uncovered branch being followed) for 19 of the cases, i.e. 65% of the time. This result provides some preliminary support to our hypothesis that simple modification strategies can successfully flip a branch condition. Figure 8-12 shows two representative examples of successful modifications. The first row shows a frontier condition in method `Stack.peek()`. In the RANDOOP-generated inputs, the method is never called on a 0-sized stack, leaving the *false* branch of the condition uncovered. The witness sequence creates a new stack, sets its size to 1, and calls `peek`. The data flow analysis returns the variable in the statement `v1=1`, used as argument to `setSize(int)`, as an influencing variable, and returns the set {`0`} as the compared-value set for the variable. The generator sets `v1` to `0`, which drives execution through the *true* branch.

The second row in the figure is an example of how a relatively complex frontier condition can be flipped using a simple modification strategy. The frontier condition is in method `Random.nextInt`, which is called from method `Collections.shuffle`. For the condition to evaluate to *true*, the input argument to `nextInt` must be a power of 2. The witness sequence creates a 100-element collection by calling `Collections.nCopies(int n,Object o)`, which returns a list consisting of `n` copies of `o`. It then calls `shuffle` on the list, which internally calls `nextInt(int)` with the size of the list (`100`) as argument. The dynamic analysis correctly identifies `v0`, the variable designating the size of the list, to be an influencing variable. Moreover, the analysis returns {`100`, `4`} as the compared-value set for `v0`: the value `4` is the result of the operation `n & -n` in the frontier condition, and is always a power of two. When the generator uses `4` as the value of `v1`, the uncovered branch in the frontier condition is covered.

frontier conditions **(*false* branch covered)**	**witness sequences** **(follow *false* branch)**	**modified sequences** **(follow *true* branch)**

public synchronized
  Object peek() {
 len = size();
 if $\boxed{\texttt{(len == 0)}}$
  throw new
   EmptyStackException();
 return elementAt(len - 1);
}
	Stack v0 = new Stack();
	int $\boxed{\texttt{v1}}$ = 1;
	v0.setSize(v1);
	v0.peek();
		Stack v0 = new Stack();
		int v1 = $\boxed{\texttt{0}}$;
		v0.setSize(v1);
		v0.peek();

public int nextInt(int n) {
 if (n<=0)
  throw new
   IllegalArgumentException
   ("n must be positive");
 	
 // if n is a power of 2
 if $\boxed{\texttt{((n \& -n) == n)}}$
  return (int)((n
   * (long)next(31)) >> 31);
 	
 int bits, val;
 do {
  bits = next(31);
  val = bits % n;
 } while(bits - val
          + (n-1) < 0);
return val;
}
	int $\boxed{\texttt{v0}}$ = 100;
	int v1 = -1;
	List v2 = Collections
	  .nCopies(v0, v1);
	Collections.shuffle(v2);
		int v0 = $\boxed{\texttt{4}}$;
		int v1 = -1;
		List v2 = Collections
		  .nCopies(v0, v1);
		Collections.shuffle(v2);

Figure 8-12: Two representative modifications performed by a branch-directed generator that led to coverage of a branch not covered by RANDOOP. (For more details see Section 8.4.)

Chapter 9

Case Studies

This chapter describes two case studies that complement the experimental evaluation described in Chapter 8. The case studies explore different uses of Randoop: the first study explores the effectiveness of *passing test cases* generated by the tool when used for regression testing, and the second study explores the effectiveness of *failing test cases*, in the context of an industrial use of the tool. The regression case study describes our use of Randoop to perform regression testing on the Java Development Kit [40] (JDK), a widely-used Java component library. We used Randoop to generate tests for Sun's 1.5 version of the JDK, and we ran the generated tests against other versions of the JDK, including a newer 1.6 beta version, and a separate implementation of the 1.5 JDK by IBM. By this process, we found many regression failures that revealed errors in all three libraries.

The second case study is an industrial case study in which a team of test engineers at Microsoft applied directed random testing to a critical component of the .NET architecture. Due to its complexity and high reliability requirements, the component had already been tested by 40 test engineers over five years, using manual testing and many automated testing techniques. Nevertheless, directed random testing revealed errors in the component that eluded previous testing, and did so two orders of magnitude faster than a typical test engineer (including time spent inspecting the results of the tool). The tool also led the test team to discover errors in other testing and analysis tools, and deficiencies in previous best-practice guidelines for manual testing.

9.1 Finding Regression Errors in the Java JDK

Our first case study involved using directed random testing to find inconsistencies between different implementations of the same API. As our subject program, we used the Java JDK. We tested three commercial implementations: Sun 1.5 [1], Sun 1.6 beta 2 [2], and IBM 1.5 [3] The goal was to discover inconsistencies between the libraries which could point to regression errors in Sun 1.6 beta 2 or compliance errors in either

[1]J2SE build 1.5.0-b64.

[2]J2SE build 1.6.0-beta2-b74.

[3]J2SE build pxi32devifx-20060124, JCL 20060120a.

```
public static void test1() {
  BigDecimal negOne = new BigDecimal(-1.0);
  BigDecimal one = negOne.divideToIntegralValue(negOne);
  BigInteger unscaled = one.unscaledValue();
  // Fails on IBM 1.5, which returns 1000000 instead of 1
  // Reveals error in IBM 1.5: it should return 1.
  assertEquals("1", unscaled.toString());
}

public static void test1() throws Exception {
  GregorianCalendar c1
    = new GregorianCalendar(1000, 1, 1);
  Date d1 = c1.getTime();
  d1.setSeconds(1);
  // Sun JDK 1.6 throws a ClassCastException;
  // this is a regression from Sun JDK 1.5.
  d1.after(d1);
}

public static void test2() {
  TreeSet ts = new TreeSet();
  ParsePosition pos = new ParsePosition(100);
  // Sun 1.6 throws ClassCastException when executing next method.
  // Reveals error in Sun 1.5 and IBM 1.5: they should also throw
  // an exception, but do not.
  ts.contains(pos);
}
```

Figure 9-1: Three inconsistencies revealed by Randoop. The first reveals an error in IBM's JDK 1.5. The second reveals a regression error in Sun's JDK 1.6. The third reveals an error in Sun's JDK 1.5 and IBM's JDK 1.5, which fail to throw an exception documents in the API documentation.

of the libraries. Recall that Randoop can optionally create a *regression oracle* for each input, which records the runtime behavior of the program under test on the input by invoking observer methods on the objects created by the input. Randoop guesses observer methods using a simple strategy: a method is an observer if all of the following hold: (i) it has no parameters, (ii) it is public and non-static, (iii) it returns values of primitive type (or `String`), and (iv) its name is `size`, `count`, `length`, `toString`, or begins with `get` or `is`.

For the experiment, we generated tests for 309 classes in the `java.sql`, `java.util`, `java.text`, `java.beans`, `java.math`, `java.security`, `java.lang`, and `javax.xml` packages. We selected these packages because execution of large amounts of randomly generated code works best for classes which do not perform many GUI and I/O operations.

We ran Randoop on Sun 1.5, using the option that creates regression oracles and the default time limit. The outcome of this was a test suite containing 41,046 regression test cases. To factor out test cases that captured non-deterministic behavior of the library, we ran the test resulting suite ten times, and used a script to remove test cases that did not consistently fail or pass—these test inputs typically included nondeterministic method calls, e.g. calls whose result depended on the current time (re-running to remove nondeterministic behavior could be easily automated). After running the test suite ten times, a total of 7098 test cases were discarded due to nondeterministic behavior.

We ran the resulting test suite using Sun 1.6 beta and a second time using IBM 1.5. A total of 25 test cases failed on Sun 1.6, and 73 test cases failed on IBM 1.5. The test suite took approximately two minutes to execute on both Sun 1.6 and IBM 1.5. On inspection, 44 out of the 98 test cases revealed inconsistencies that uncovered 12 distinct errors in the implementations (other inconsistencies reflected different implementations of a permissive specification). Interestingly, we found errors not only in Sun 1.6 (2 errors) and IBM 1.5 (8 errors), but also in Sun 1.5 (5 errors). For example, one inconsistency between IBM 1.5 and Sun 1.5 revealed an error in Sun 1.5, despite the fact that the test case failed on IBM 1.5. Figure 9-1 shows three inconsistencies revealed by the test suite. For more specific inconsistencies, see [80].

All distributed JDKs must pass an extensive compliance test suite [56], regrettably not available to the public nor to us. Nevertheless, RANDOOP was able to find errors undiscovered by that suite. Internally, IBM extensively uses comparisons against the Sun JDK during testing, but they estimate that it will take 100 person-years to complete that comparative testing [49]. A tool like RANDOOP could provide automated support in such a process.

9.2 Industrial Case Study: Finding Errors in .NET

The real assessment of a test generation technique's effectiveness is its performance in the real world. In an industrial setting, testing techniques are used under a very different set of constraints from a research setting. Practicing test engineers have tight deadlines and large amounts of code to test. For an automated test generation tool to succeed in this environment, it must reveal errors important enough to be fixed and it must reveal these errors in a cost-effective way, taking up less human time than manual testing or existing automated techniques. These qualities can be particularly difficult to measure in a research setting.

This section describes an industrial case study for directed random testing that sheds light on its effectiveness when used by test engineers in an industrial testing environment, and in comparison with the application of other test generation techniques already in use by the engineers. A test team at Microsoft applied directed random test generation to a large component of the .NET Framework [74] used by thousands of developers and millions of end users. The component sits low in the .NET framework stack, and many .NET applications depend on it for their execution. For this reason, the component has had approximately 40 testers devoted to testing it over a period of five years. It has undergone manual unit, system, and partition testing, as well as automated testing including fuzz, robustness, stress, and symbolic execution-based testing. Because of proprietary concerns, we cannot identify the .NET component analyzed. We will refer to it as "the component" from here on.

The test team's knowledge of the average human effort required to manually find an error in the component under test allowed us to quantify the benefit of directed random testing compared to manual testing. Since the team has applied many testing techniques to the component, we were also able to learn about the effectiveness of directed random testing compared with these techniques.

A summary of the study's results:

- Directed random test generation found more errors in 15 hours of human effort and 150 hours of CPU time than a test engineer typically finds in one year on code of the quality of the component under test. The technique found non-trivial errors, including errors that arise from highly specific sequences of operations. Moreover, these errors were missed by manual testing and by all previously-applied automated test generation techniques. Based on these results, the tool implementing directed random testing was added to a list of tools that other test teams at Microsoft are encouraged to use to improve the quality of their testing efforts.

- As a result of applying directed random testing to the component, the test team found and fixed errors in other automated testing tools, performed further manual testing on areas of the code that the technique showed to be insufficiently tested, and implemented new best practices for future manual testing efforts. In other words, the technique was used beyond bug finding, as an assessment tool for the test team's existing testing methodologies, and spurred more testing activities.

- After a highly productive error detection period, directed random testing plateaued and eventually stopped finding new errors, despite using different random seeds. This observation mirrors the results of a recent, unrelated study of random testing [44]. We provide a tentative explanation of the plateau effect for the case of directed random testing applied to the .NET component, and propose research directions to address the effect.

The rest of the section is organized as follows. Section 9.2.1 gives an overview of the .NET component under test. Section 9.2.2 describes the process that the test team used in applying directed random test generation to the component. Section 9.2.3 discusses the results, including the number and type of errors revealed, the reason why other techniques missed these errors, and challenges that directed random testing faces in finding more errors over time.

9.2.1 Overview of .NET Component

The software used in this study is a core component of the .NET Framework [74]. It implements part of the functionality that allows managed code (code written in a high-level programming language like C#) to execute under a virtual machine environment. The component is required for any .NET application to execute. It is more than 100KLOC in size, written in C# and C++, and it exports its functionality in an

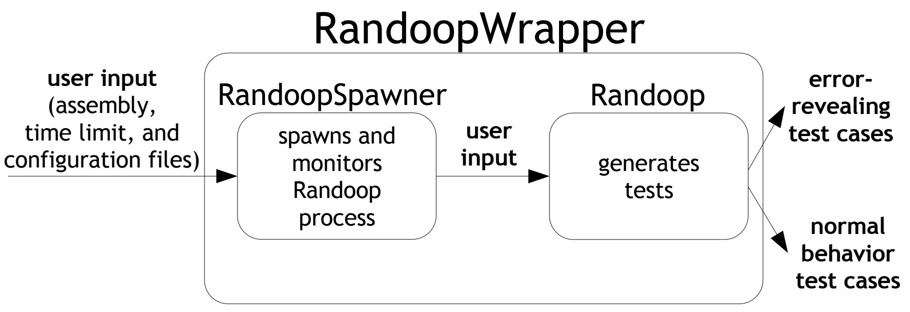
RandoopWrapper
user input (assembly, time limit, and configuration files)
RandoopSpawner
spawns and monitors Randoop process
user input
Randoop
generates tests
error-revealing test cases
normal behavior test cases

Figure 9-2: RandoopWrapper is a wrapper process around Randoop that makes it more robust to crashes caused by arbitrary code execution. First, RandoopSpawner spawns a Randoop process. If Randoop crashes, RandoopSpawner spawns a new Randoop process with a new random seed. This continues until the user-specified time limit expires.

API available to programmers both inside and outside Microsoft. Many applications written at Microsoft use the component, including the BCL (a library for I/O, graphics, database operations, XML manipulation, etc.), ASP.NET (web services and web forms), Windows Forms, SQL, and Exchange.

The software component has undergone approximately 200 man years of testing, not counting developer testing (most developers write unit tests). The test team has large computational resources for testing, including a cluster of several hundred machines. The test team has tested the component using many techniques and tools. In addition to manual (unit and system) testing, the team has developed tools for performance, robustness, stress, and fuzz [34] testing. They have created tools that automatically test code for typical corner cases and values, such as uses of `null`, empty containers or arrays, etc. Additionally, thousands of developers and testers inside Microsoft have used pre-release versions of the component in their own projects and reported errors. To facilitate testing, the developers of the component make heavy use of assertions.

At this point, the component is mature and highly reliable. A dedicated test engineer working with existing methodologies and tools finds on average about 20 new errors *per year*. During the earlier years of its development cycle, this figure was much higher. One of the goals of this study was to determine if directed random testing could find errors not found by previous testing techniques, on software of the maturity level of the component.

9.2.2 Process

To aid the case study, and based on discussions with the test team, we made two modifications to Randoop.

- **Robustness.** Recall that to improve efficiency, the tool executes method sequences in the same process where it executes its own code, using .NET's reflection infrastructure. In early runs of Randoop on the component under study, some method sequences caused the tool to be forcefully terminated by the operating system, due to execution of sequences containing methods that attempted to perform a low-level OS operation for which Randoop had no privileges. Thus, Randoop sometimes terminated before the user-specified time limit. The test team wanted to be able to run the tool for long periods of time. To improve Randoop's robustness, we wrapped Randoop in another program, RandoopWrapper (Figure 9-2). RandoopWrapper takes the user's input, spawns a Randoop process on the given input, and monitors the execution of the Randoop process. If Randoop terminates before the user-specified time limit is reached, RandoopWrapper spawns a new Randoop process, using a new random seed. Method sequences that lead to Randoop crashes are also output as potentially error-revealing test cases.

- **Oracle.** The test team was interested in using Randoop to check for three kinds of behavior: assertion violations, access violations, and unexpected program termination. The reason is that these three kinds of behaviors in the component under test are nearly always errors. To check for unexpected program termination, we modified Randoop to always output a test input as source code to a file before executing it. Unexpected termination was detected by RandoopWrapper, which could report the name of the file containing the responsible test input. To check for assertion and access violation, we modified Randoop directly. (Randoop for .NET does not implement contracts via interfaces; if it did, we could have created contracts that check for these behaviors without modifying the tool.)

To conduct the study, we gave a copy of Randoop to the test team, along with instructions on how to run the tool. Since the tool is fully automatic, works directly on assemblies, and outputs compilable, error-revealing test cases, the test team had no trouble understanding its purpose. They started using Randoop with its default settings: one minute generation time limit and no configuration files. Object-equality pruning (Section 7.4) did not improve performance for the component under test, and was not used in the case study.

As they discovered errors, the test team created error reports and assigned engineers to fix them. It was not always possible to fix the code immediately, so the test team altered Randoop's configuration files to instruct it not to explore methods that led to error-revealing behavior. This prevented Randoop from rediscovering the same error in subsequent runs. As they became more familiar with the tool, they used the tool in more sophisticated ways, creating different configuration files to focus on different parts of the component.

We met with the test team on a regular basis to discuss their experience with the tool, including the time they had spent using it and inspecting its results, as well as its effectiveness compared with their existing manual test suites and testing tools. Based on requests by the test team, we also implemented a number of new configurable options, such as the ability to output all sequences that Randoop generated,

total number of tests generated	4,000,000
distinct errors revealed by Randoop	30
total CPU time required to reveal the errors	150 hours
total human time spent interacting with Randoop	15 hours
average errors revealed by a tester in 1 year of testing	20

Figure 9-3: Case study statistics.

regardless of their classification (Section 9.2.4 discusses the way that the test team used this option).

9.2.3 Cost Effectiveness

Figure 9-3 summarizes the results of the test team's effort. Randoop revealed 30 serious, previously unknown errors in 15 hours of human effort (spread among several days) and 150 hours of CPU time. Each error was entered as a bug report (many have since been fixed). The 15 hours of human effort included inspecting the error-revealing tests output by Randoop. To place the results numbers in context, recall that for a code base of the component's level of maturity, a test engineer will find approximately 20 errors per year.

The kinds of behaviors that the tool currently checks for (assertion violations, access violations, and unexpected termination) are almost always indicative of errors in the code, so false positives were not as much a problem as *redundant tests:* test that were syntactically distinct but revealed the same error in the implementation. The hours reported include time spent inspecting and discarding redundant tests.

> *In terms of human effort, a test engineer using Randoop revealed more errors in 15 hours than he would be expected to find in a year using previous testing methodologies and tools.*

9.2.4 Error Characteristics

This section present the observed characteristics of the errors that Randoop found, and representatives examples. Each section presents an observation followed by examples.

Errors in well-tested code

Randoop revealed errors in code on which previous tests had achieved full block and arc coverage. An example is an error dealing with memory management and native code. The component code base is a combination of memory-managed (garbage collected) code as well as native code with explicit memory allocation. When native code manipulates references from managed objects, it must inform the garbage collector of any changes (new references, references that can be garbage-collected, etc.).

Randoop created a test input that caused an internal portion of the component to follow a previously untested path through a method. This path caused the native code to erroneously report a previously used local variable as containing a new reference to a managed object. In this specific path, the address of the reference was an illegal address for a managed object (less than 32k but greater than 0). In a checked build (the version of the component used during testing, which includes assertion-checking code), the component checks for the legality of the addresses, and threw an assertion violation stating that a bad value was given as a reference into the garbage-collected heap.

The erroneous code was in a method for which existing tests achieved 100% block coverage and 100% arc coverage. After fixing the error, the test team added a new regression test and also reviewed (and added test cases) for similar methods. This is an example of an error discovered by Randoop that led to testing for more errors of the same kind, reviewing existing code, and adding new tests.

> *Directed test generation revealed errors in code in which existing tests achieved 100% code coverage.*

Using Randoop's output as input to other tools

At the beginning of the study, we expected Randoop to be used as an end-to-end tool. However, the test team started using Randoop's test inputs (which are standalone executable files) as input to other tools, getting more functionality from each generated test. The test team requested that we add an execution to Randoop in which it outputs all the test inputs it creates, even if they were not error-revealing. Their goal was to use other tools to execute the inputs under different environments in order to discover new errors.

Among the tools that they used were stress and concurrency testers. An example is a tool that invokes the component's garbage collector after every few instructions, or a tool that runs several Randoop-generated test inputs in a stress tool that executes a single tests input multiple times in parallel (with a separate thread executing the same input). This process led the test team to discover more errors. Using the latter tool, the test team discovered a race condition that was due to incorrect locking of a shared resource. The error was revealed only after a specific sequence of actions by a method, involving locking an object, performing an operation, and finally calling another method that reset the state of the thread. The team fixed the error in the method that reset the thread state, implemented a tighter protocol around the specific behavior, and did a review of similar constructs (the review found no other issues).

> *The test team used* Randoop*'s generated tests as input to other testing tools, increasing the scope of the exploration and the types of errors revealed beyond those that Randoop could find.*

Testing the test tools

In addition to finding errors directly in the component, Randoop led the test team to discover errors in their existing testing and program analysis tools. An example of this is an error in a static analysis tool that involved a missing string resource. In the component, most user-visible strings (for example, exception messages) are stored in a text file called a resource file. The resource file is included with the product binary at build time, and is accessed when a string is needed during execution. This approach simplifies language localization.

The test team had previously built a simple analysis tool to detect unused or missing resource strings. The tool inspects the component source code and checks that each resource is referenced at least once in the code, and that the resource exists. However, the tool had a bug and it failed to detect some missing strings. Randoop generated a test input that caused an infrequently-used exception type to be raised. When the virtual machine looked up the exception message string, it did not find the string and, in the checked build, led to a fatal assertion violation. On a retail build (the version of the component shipped to customers), the missing string produced a meaningless exception message.

After adding back the missing resource, the test team fixed the error in their resource checking tool and did further manual testing on the tool to verify that it worked properly.

> *In addition to revealing errors in the .NET component, Randoop revealed errors in the test team's testing and program analysis tools.*

Corner cases and further testing

For software of high complexity, it is difficult for a team of testers to partition the input space in a way that ensures that all important cases will be covered. While Randoop makes no guarantees about covering all relevant partitions, its randomization strategy led it to create test cases for which no manual tests were written. As a result, directed random testing discovered many missed corner cases.

The knowledge gained from the discovery of the corner cases led the test team to consider new corner cases, write new tests, and find more errors. In some cases, the discovery of a new error led the test team to augment an exiting testing tool with new checks for similar corner cases, and in other cases the discovery of an error led the test team to adopt new practices for manual testing.

The first example is an error that uncovered a lack of testing for empty arrays. The component has a container class that accepts an array as input to initialize the contents of the container. The initialization code checks the legality of the input data by iterating over the array and checking that each element is a legal element for the container. An empty array is legal. One of access methods expected did not handle the case in which the input array is empty. In this case, the method incorrectly assumed that it was an array of bytes and started reading bytes starting from the base address of the array. In most cases, this would quickly lead to a failure due to malformed data, and in other cases (one created by Randoop), the method would fail with an access violation. The test team fixed the error, reviewed other access methods, and as a result fixed other similar issues. As a result of this error, the team updated their "best practices" to include empty arrays as an important input to test.

This area of the component contained a large number of tests for different kinds of initialization arrays. However, the sheer size of the state space made it impossible to test all possible combinations of inputs, and the manual tests were incomplete.

The second example is an error that uncovered a lack of testing for I/O streams that have been closed. When an I/O stream is closed, subsequent operations on the stream should fail. A Randoop-generated test showed that calling a successive set of state-manipulating methods on a closed stream would lead to one of the operations succeeding. In the specific case, Randoop generated a call sequence that would create a specific stream, do some operations and then close the underlying stream. The component has many test cases that test for similar behaviors, i.e. testing that operations on closed streams fail in specific ways. Again, due to the size of the component, some important cases were missing. In a checked build the test case caused an assertion violation, and on a retail build it led to being able to access certain parts of the stream after it is closed. The error has been fixed, test cases have been added, and reviews of similar code have been completed.

> *The errors that Randoop revealed led to further testing activities unrelated to the initial random testing effort, including writing new manual tests and adopting new practices for manual testing.*

9.2.5 Comparison with Other Test Generation Techniques

The errors that Randoop revealed were not revealed using the team's existing methodologies and tools, including a very large collection of manually-written unit and system tests, partition testing, fuzz testing, and program analysis tools like the one described in Section 9.2.4. Conversely, there were many errors revealed by previous efforts not revealed by Randoop. In other words, Randoop was not subsumed by, and did not subsume, other techniques.

Randoop vs. Non-Random Test Generation

According to the test team, a major disadvantage of Randoop in comparison with manual and non-random automated techniques is Randoop's lack of a meaningful stopping criterion. After several hours of running Randoop without the tool producing a new error-revealing test case, they did not know whether Randoop had essentially exhausted its power and was "done" finding all the errors that it would find, or whether more computer resources would lead to new errors. For example, towards the end of the study, the test team ran Randoop for many hours on several dedicated machines but the tool did not reveal any new errors. Other techniques have more clear-cut stopping criteria. A static analysis tool terminates when the analysis is complete; symbolic execution based techniques terminate when they have attempted to cover all feasible paths, etc.

The experiences of the test team suggests that RANDOOP enjoys two main benefits compared with non-random automated test generation approaches. One is its scalability. For example, concurrently with using RANDOOP, the test team used a new test generator that outputs tests similar to RANDOOP, but uses symbolic execution [62], a technique that instruments the code under test to collect path constraints, and attempts to solve the constraints in order to yield test inputs that exercise specific branches. The symbolic execution tool was not able to find errors in the component during the case study. One of the reasons is that the tool depends on a constraint solver to generate tests that cover specific code paths, and the solver slowed down the tool and was not always powerful enough to generate test cases. In the amount of time it took the symbolic execution tool to generate a single test, RANDOOP was able to generate many test cases.

The second main benefit is that RANDOOP's (and more generally, random testing's) randomized search strategy protects test engineers against human bias. For example, in Section 9.2.4 we discuss that the test team had not previously considered testing a set of methods using empty arrays, and RANDOOP revealed an error elicited via an empty array. The test team had previously omitted empty arrays from testing because the engineer that crafted the test cases for the method in question did not consider empty arrays an interesting test case at the time.

Human bias is not limited to manual testing; automated tools can suffer from the biases of their creators. For example, the test team has created automated testing tools that test methods that take multiple input parameters, using all possible combinations of a small set of inputs for each parameter slot. RANDOOP revealed errors that were not revealed using the inputs programmed into the tool.

> RANDOOP*'s randomized search revealed errors that manual and non-random automated techniques missed because of human bias or lack of scalability. However, Randoop has no clear stopping criterion, which makes it difficult to gauge when the tool has exhausted its effectiveness.*

Randoop vs. Fuzz Testing

Previous to RANDOOP, the component had undergone extensive fuzz testing [34] (a random testing technique that generates data streams or files containing random bits or characters) on nearly every format and protocol of the component. Like directed random testing, fuzz testing is also unbiased, but previous fuzzing efforts did not reveal the errors that RANDOOP revealed.

A reason for this is that fuzz testing has been traditionally done on data-intensive software that take as inputs files, network packets, etc. Fuzzing is less frequently applied to domains that deal with both data and control, such as method sequences in object-oriented libraries. The errors that RANDOOP found turned out to be about both data (the input to methods) and about control (the specific sequence of methods). In order to discover data errors, some amount of control structure was necessary (a sequence of method calls), and in order to discover control errors, some data was necessary (inputs to methods). RANDOOP helped bridge the divide between data and control.

Fuzzing is effective in generating test inputs that explore either data or control. When the structure of inputs includes both data and control, directed method sequence generation can be more effective.

9.2.6 The Plateau Effect

The errors revealed by Randoop did not emerge at a uniform rate during the testing process. Instead, the rate of error discovery was quick at first and then decreased. During the first two hours of use, Randoop revealed 10 distinct errors (5 errors per hour). Then the error-finding rate fell to 2 errors per hour for approximately 10 hours. After that, Randoop ceased to output error-revealing test cases.

To make Randoop more effective, the test team tried different strategies, such as creating different configuration files that targeted specific portions of the component. These efforts revealed more errors, but did not alter the trend towards diminishing returns for the effort expended. Towards the end of the case study, the test team switched from running Randoop on a desktop to running it on parallel in a cluster of several hundred machines, using different combinations of random seeds and configurations. These runs revealed fewer errors than the initial runs on a single desktop.

Groce et al. [44] also observed this effect using a technique similar to Randoop's to generate tests consisting of sequences of file system operations for flight software.

After an initial period of effectiveness, directed random test generation yielded diminishing results.

Chapter 10

Discussion

This chapter discusses the strengths, limitations, and future research directions for directed random testing, drawing from the experimental results and case studies we performed to assess its effectiveness, as well as from our and other users' experience using RANDOOP. The greatest strengths of the directed random testing approach are its *error-revealing effectiveness* (as evidenced by the many errors revealed by RANDOOP in our experiments and case studies) and its *cost effectiveness*, as evidenced by the hundred-fold productivity gains by Microsoft engineers using the tool over 15 hours of human effort. Combined, these strengths make the technique an attractive point in the space of test generation techniques, yielding useful test cases in a short period of time.

The limitations of the technique mainly stem from the fact that directed random testing is primarily a random test generation technique. As such, it does not guarantee code or input space coverage and has no clear stopping criterion, which can make it difficult to determine how many resources (human or computational) to devote to the technique. A second deficiency of the technique is that it is not well-suited to unit testing internal programs units (e.g. private methods), because such units often require complex setup steps (e.g. reading a file) unlikely to be achieved using randomly-generated sequences of operations. The technique is better-suited to testing reusable component libraries that require less specific setup to use. Finally, directed random testing can fail to cover some code that requires highly-specific inputs (for instance, specific combinations of numeric inputs) to be exercised. We also propose potential research directions to overcome, or at least mitigate, some of these deficiencies.

10.1 Strengths of Directed Random Testing

We can categorize the strengths of directed random testing in terms of two aspects: error-revealing effectiveness and cost-effectiveness.

10.1.1 Error-Revealing Effectiveness

Our experimental results and case studies demonstrate that directed random testing is effective in revealing critical errors in large, widely-deployed, and heavily-tested applications. On a collection of 14 widely-used software libraries, directed random testing revealed many previously unknown errors (Section 8.2). After we discovered the JDK errors, we submitted some of the errors to the Sun bug database, but did not receive any notice about the status of our bug reports. An indication that many of these errors are real is the fact that the errors for which we published failing test cases in [82] were fixed in the 1.6 version of the JDK. Directed random testing also revealed problematic regression differences between different versions of the JDK (Section 9), which could cause an application that depends on the old behavior of the libraries to malfunction. The above results were achieved by our own use of the technique, raising a valid question: would the technique be as effective when used by someone other than the technique's creators? Our industrial case study (Section 9.2) answers this question in the affirmative, and shows that the technique revealed critical errors in a mature component when used by a team of test engineers.

Compared with other other automated test generation techniques applied to the same software artifacts under test, our results indicate that directed random testing can be more effective in revealing errors and achieving code coverage than other systematic and random approaches. In our experiments, Randoop typically revealed more errors than undirected random testing, model checking, and symbolic execution, and in the Microsoft case study, Randoop revealed more errors during the same period of time as a technique based on symbolic execution that, in principle, can yield greater coverage.

While the results are exciting and are strong indicators of the technique's promise, it would be incorrect to conclude that directed random testing *subsumes* other techniques; this is simply not the case. A more objective statement is that directed random testing appears to outperform other automated test generation techniques on software artifacts that declare a large number of operations, and characterized by operation trees with very large branching factor. This characteristic is shared by all the subject libraries we used to evaluate the technique. For example, Figure 10-1 shows the size of the input space for the libraries from Section 8.2, all of which declare hundreds of operations (from 140 to 2412) and have large branching factors (from 80 to 25,000). Gray cells in the table represent sequence lengths for which generating all sequences at the given length would take more than one month of generation time, assuming a generator that creates and executes 1,000 operation sequences per second (a reasonable number based on our experiments). For all the libraries, generating all sequences of length 5 would take several months of effort; for some it would take much longer than that to generate all sequences of even length 3 or 4. In other words, software artifacts of this size are challenging for exhaustive techniques, which spend the vast majority of their time exhaustively sampling a dense, but tiny and localized, portion of the input space. Directed random testing explores the space more effectively not because it explores a larger portion of the input space—it only explores a tiny fraction of the space as well—but rather because instead of a dense,

sequence length	1	2	3	4	5
java.util	1.6 E3	2.7 E6	4.6 E9	7.8 E12	1.3 E16
javax.xml	3.8 E1	2.2 E3	1.7 E5	1.5 E7	1.3 E9
chain	2.8 E1	1.3 E3	9.7 E4	1.0 E7	1.0 E9
collections	6.9 E3	4.9 E7	3.4 E8	2.4 E15	1.7 E19
jelly	9.1 E2	1.5 E6	3.5 E9	8.1 E12	1.8 E16
logging	3.5 E1	1.6 E3	1.1 E5	1.1 E7	1.0 E9
math	2.4 E4	6.2 E8	1.5 E13	3.8 E17	9.6 E21
primitives	3.7 E2	1.5 E5	6.3 E7	2.6 E10	1.0 E12

Figure 10-1: Size of the input space for Java libraries. Shaded cells represent input depths that would require more than one month of computer time to explore exhaustively, assuming 1,000 inputs generated per second.

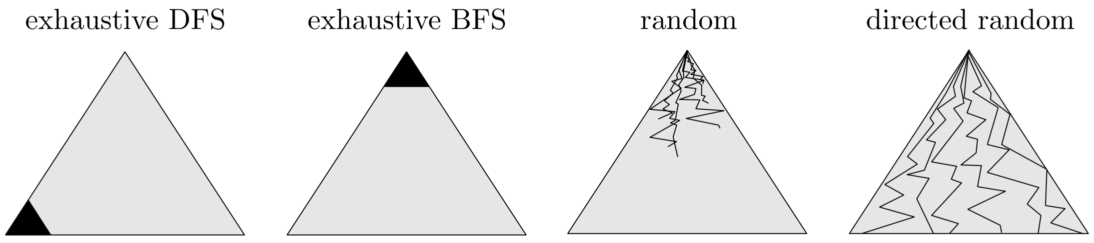

Figure 10-2: Graphical depiction of the nature of the portions of the input space explored by four different input generation approaches.

local sampling, the technique performs a sparse, but more diverse sampling of the space. Our results suggest that for large libraries, sparse, global sampling can reveal errors more efficiently than dense, local sampling. Figure 10-2 shows an informal graphical depiction of the exploration strategy of different techniques. Undirected random generation is less effective because it is unable to reach very deep into the operation tree (where the density of errors is higher when doing random generation), due to software malfunctions from illegal inputs.

The test engineers at Microsoft that participated in the case study from Section 9.2 reached the same conclusion after using Randoop and comparing it with other approaches. As an engineer put it, Randoop was effective revealing errors in the .NET component, which has with a "large surface area" (i.e. a large number of methods) because it did not get stuck on a small portion of the input space, and was also superior than undirected random testing tools because it performed a "smart" version of API fuzzing.

Our results (Section 8.1) also suggest that directed random testing achieves coverage comparable with systematic testing techniques when applied to single classes, e.g. integer container data structures like binary trees, red-black trees, etc. However, it is not clear that when testing a single class, or a small set of classes, directed random testing is preferable than systematic techniques. While our experiments show that directed random testing achieves code coverage on par with systematic techniques, it has an important disadvantage: it cannot guarantee that all inputs have been explored, a guarantee that can be achieved in practice with systematic techniques, when applied to a small and focused portion of code. This guarantee becomes important if the code under test is a crucial portion of a larger software system. For example, the same Microsoft engineers that found Randoop to be effective in generating tests for large component libraries also mentioned that they would be willing to devote a significant amount of time and resources using an exhaustive approach that guaranteed full input coverage (within some bound) when the target was a small class or a set of classes that required very high reliability.

10.1.2 Cost Effectiveness

The second principal strength of directed random testing is its cost effectiveness, as evidenced by the experience of the Microsoft test team, which was able to do one man-year's worth of error discovery in 15 hours of effort (Section 9.2). The technique's cost effectiveness was no accident: one of the principal goals in developing the techniques encompassing directed random testing was to produce a tool that would give useful results with minimal user effort. This goal is reflected in most aspects of the technique:

- At the input level, we wanted to preserve the lack of user requirements that makes random testing cost effective in terms of the time required creating input artifacts. For this reason, we developed heuristic pruning, a technique that automates the determination of illegal and redundant inputs. Our experimental results show that the heuristics are able to guide the search away from such inputs. The results also show that the heuristics are far from perfect in discarding redundancy: in our application of Randoop to Java and .NET libraries (Section 8.2), almost every error that Randoop revealed was revealed by several (redundant) inputs. However, the heuristics contributed to Randoop revealing more errors than several other forms of random testing that do not use heuristics, including Randoop without pruning heuristics, a random walk, and JCrasher.

- The largely-random nature of directed random testing, and the heuristics it employs based on runtime behavior of the software, yield a technique that can be implemented without requiring code instrumentation, constraint solvers, or a specialized virtual machine, which systematic techniques based on model checking or symbolic execution often require. This makes the technique immediately applicable to most code artifacts; for example, we were able to immediately run Randoop on the .NET Framework libraries and three industrial implementations of the JDK, and found previously-unknown errors. The lack of complex machinery makes the technique cost effective because it does not require the engineer to work around deficiencies, errors, or heavy-weight technology in the testing tool itself. For example, one of the reasons the Microsoft test team (Section 9.2) was less successful in finding errors with the symbolic-execution-based test generation tool that they were using concurrently with Randoop was because it used a specialized virtual machine to execute .NET code, and the virtual machine at the time did not handle aspects of the language, e.g. some native calls. On the other hand, because Randoop executes code directly, it never had any trouble making native calls, and was immediately applicable to the entire component under test.

- At the output level, we wanted a technique that would output test cases that could immediately be used by a test engineer. This is one of the primary reasons why the technique maintains a component set of sequences, as opposed to a component set of values: a sequence can be output as a sequence of operation calls, and is trivial to convert to a compilable unit test. Programmers using Randoop immediately understood the output of the tool and its purpose, and were able to add the generated test cases to their test suite with little or no modification.

10.2 Limitations of Directed Random Testing

Our experiences and those of other users of directed random testing point out a number of limitations of the technique. Most of these limitation are common to any approach based on random testing.

10.2.1 Lack of Stopping Criterion

A fundamental limitation of directed random testing is its lack of a clear stopping criterion. This limitation can be particularly important in an industrial setting where a test team must plan how many resources to devote to a technique. (Not surprisingly, the lack of stopping criterion was one of the main criticisms the Microsoft engineers had about Randoop in the industrial case study.) A partial solution is to stop when the tool reaches a coverage plateau: the tool can track coverage during the generation, and stop generating inputs when coverage stops increasing for a long period of time. The stopping criterion seems reasonable, but it fails to guarantee that running the tool for a longer period will not reveal a new, potentially-critical error.

10.2.2 Internal Components

A technique that generates arbitrary sequences of operations is well-suited for testing libraries of reusable components with clear, modular interfaces and little or no internal global state. On the other hand, it is ill-suited for testing internal components of programs that are intended for use in highly specific situations, or that require specific steps to be performed before the components can be used. To assess the effectiveness of the technique when applied to the internal components of an end-to-end program, we asked a developer of Daikon [31] (a dynamic analysis tool) to use Randoop to generate test cases for some internal classes in the program. Randoop was unable to generate useful test cases for the classes, because the classes (1) were intended to be used at very specific points during the execution of Daikon, (2) referred to static state in other Daikon classes that required reading an input file to be properly set up, and (3) were not implemented as reusable components. The deficiency is not unique to random testing; using a model checker like JPF [91] or a concolic testing tool like jCUTE [86] to generate arbitrary method sequences for internal Daikon classes would also lead to largely useless test inputs.

A way to partially overcome this deficiency is for the test engineer to create "helper" classes. The helper classes can declare methods that perform any required setup before the classes can be used, or methods that construct and return well-formed objects. The helper classes can be given as input to the generation tool, along with the classes under test. Another potential new approach, which we have begun exploring [8], is to use an example execution of the program to infer a model of legal uses of the internal classes, e.g. a model that includes sequences of operation that are required to put objects of the classes under test in legal states.

10.2.3 Unreachable Code

On realistic-sized libraries, Randoop rarely achieves 100% code coverage. For example, in the large-scale experiment detailed in Section 8.3, the branch code coverage achieved by the technique, computed across hundreds of runs of the technique, did not go beyond 80% for some of the subject libraries, and was closer to 50% for most. Regardless of how long the technique runs, some branches appear out of reach of the technique. Based on an analyzing the coverage achieved by Randoop on the artifacts we used for evaluation, and talking to the Microsoft engineers that inspected Randoop's output, we discovered two primary reasons for the technique's inability to reach some code.

Challenging Constraints

Some branch conditions express constraints that are challenging for a random-based technique to satisfy. Section 6.2 describes branch-directed generation, a technique that aims to cover branches left uncovered by random test generation. Our preliminary evaluation of the technique suggests that it is promising, especially for covering branches containing numerical or pointer comparisons. However, our manual inspection of uncovered branches indicate that many constraints are of a more complex structural nature, e.g. a branch that is followed if a list contains an element that is structurally equivalent to an element already in the list. To cover such branches, it may not be sufficient to modify the primitive inputs appearing in an operation sequence, or replace uses of a variable by another variable, but rather to determine a sequence of operations that create the appropriate structure that causes the branch to be followed.

A potential new approach to this problem is to combine representation-based generation techniques (like TestEra [61] and Korat [16]) that create values directly, without using a set of operations, with operation-level techniques like directed random testing. For example, Korat [16] generates data structures that satisfy a condition (typically a representation invariant) specified by the user, using the invariant to guide the generation. Using an approach like Korat, specifying conditions required to cover a specific branch, could result in a more direct way to obtain inputs than trying to assemble a sequence of operations. A hybrid approach that combines operation-level and representation-level generation could yield most of the automation benefit of the former and the ability to generate structurally-complex inputs of the latter.

Distribution Unfairness

Given a software artifact composed of multiple components, a reasonable requirement for a test generation tool is that it be *fair,* meaning that it distribute its computational resources fairly among the different components. When analyzing Randoop's output, we discovered that Randoop is not fair. Randoop selects which method to test next uniformly at random from among the set of all public methods in the assembly under test (the same applies to constructors). This strategy can lead to some classes being explored (and covered) more than others. A class that defines five constructors will be explored more heavily than a class that defines one constructor. A class that defines a nullary constructor (a constructor that requires no arguments) will be explored more heavily than a class whose constructor requires an object of a type that is difficult to create.

Randoop tends to focus the generation effort on classes that declare nullary constructors or that define several constructors, at the expense of classes that require fewer constructors or constructors that require more complex setup. Because Randoop is incremental and creates new method sequences from previously-created sequences, the initial favoring of a few classes can lead to a feedback loop in which classes that were initially easier to create are tested more heavily in later stages of generation.

A way to mitigate distribution unfairness may be to use a distribution other than Randoop's uniform random distribution. In fact, it may be desirable to use an *adaptive distribution.* At the start of generation, Randoop could maintain a mapping from each constructor and method to a "weight" number indicating the relative probability that the given method or constructor will be selected when creating a new sequence (members with higher weight have a higher change of being selected). At the beginning, all methods and constructors would have equal weight. At regular intervals, Randoop would update the weights: members for which more sequences have been created would be given lower weight, and members for which fewer sequences have been created would be given higher weight. Thus for example, a class with several constructors or with a unary constructor, for which Randoop quickly generates many inputs, would eventually receive a lower weight, which would increase the chances that the tool explores other, less well-explored classes.

Chapter 11

Related Work

This chapter surveys previous work related to directed random testing. After briefly surveying elements of software testing related to our work, we discuss the history of random testing, from work dating back to the seventies on random testing of compilers to Miller's seminal work on fuzz testing and recent work on testing object-oriented programs. We also survey the techniques in the exhaustive testing domain closest to our own work, including techniques that exhaustively generate operation sequences and techniques that exhaustively generate all (non-equivalent) structured inputs such as objects. There have been a number of published comparisons of random and systematic techniques; we discuss the results. Finally, we discuss recent hybrid techniques that combine elements of systematic and random test generation.

11.1 Software Testing

A good modern introduction to software testing is the recent book by Amman and Offutt [3]. Their book focuses largely on the different coverage criteria that a test engineer can use as a guide to the manual creation of test cases. Older books in the field also devote most of their content to coverage criteria and manual test generation, including Bezier's [12] and Myers [73].

Recently, the automated test generation field has experienced a resurgence, and most relevant to this thesis, many research techniques have been proposed that combined the random test generation with non-random techniques based on symbolic execution. Section 11.3 gives an overview of the techniques most related to directed random testing.

Kaner's 2003 paper "What Is a Good Test Case?" [59] proposes a list of possible objectives for writing and running tests. The objectives of running test cases are not always technical: in addition to finding defects, blocking premature product releases, and assessing conformance to specifications, a software development organization may run test cases to minimize technical support costs, avoid safety-related lawsuits, or to adhere to government regulations. Kaner also proposes a list of objectives that "good" test cases must have, which we used as a starting point for our discussion of qualities of a good test case. Our list of qualities touches essentially all the objectives proposed by Kaner except helping the test engineer develop insight into the product, which we consider helpful but outside the scope of the thesis.

Jackson et al. argue in a recent National Academies report that software testing alone is rarely sufficient for showing that a system is dependable, and should be augmented by other kinds of analysis when making a case for a software artifact's dependability to a certifier or regulator [55].

11.2 Random Testing

Work on random test generation dates back to a 1970 paper by Hanford. In it he reports on the "syntax machine," a program that generated random but syntactically-correct programs for testing PL/I compilers [48]. The next published work on random testing was a 1983 paper by Bird and Muñoz [14]. They describe a technique to generate randomly-generated "self-checking" test cases: test cases that in addition to generating test inputs generated checks for expected outputs using knowledge of the semantics of the software under test. For example, when generating random programs for testing a compiler, the generator kept track of the expected values of arithmetic computations and inserted the appropriate checks into the test case. They also applied their technique to checking sorting and merging procedures and graphic display software.

Since Bird and Muñoz's work, many random test generation techniques have been developed to test a wide range of software systems; a partial list includes parsers, compilers, stream utilities common in Unix-like operating systems, file system and other operating system components, graphical user interfaces, functional programs, and object-oriented programs. The detailed of the techniques developed are as diverse as the systems they have been applied to, but all essentially generate inputs randomly from a predefined input grammar, without analyzing the underlying implementation of the software under test or a formal specification.

One of the best-known works in the field is Miller et al.'s "fuzz testing" paper, where they generate random ASCII character streams and use them as input to test Unix utilities for abnormal termination or non-terminating behavior [72]. In subsequent work, they extended fuzz testing to generate sequences of keyboard and mouse events, and found errors in applications running in X Windows, Windows NT and Mac OS X [34, 71]. Today, fuzz testing is used routinely in industry. It is frequently used as a tool for finding security vulnerabilities and is applied to test formats and protocols that attackers might use to gain unauthorized access to computers over a network. Other work that applies random test generation to operating systems code includes Kropp's use of random test generation to test low-level system calls [63], and Groce et al.'s use of random test generation at NASA to test a file system used in space missions [44]. In all these studies, a random test generator invariable found many errors in the software under test, and many errors were critical in nature. For example, Groce et al. found dozens of errors in the file system under test, many of which could have jeopardized the success of a mission.

Claessen and Hughes developed QuickCheck, a tool that generates random test inputs to check user-defined correctness properties of programs written in Haskell [20]. The tool also lets the use define custom test data generators for specific types. They report on their experience using the tool, and conclude that random input generation is effective in testing functional code.

JCrasher, Rostra, Symstra and Eclat use Henkel and Diwan's term-based representation [50]. As we demonstrate in Chapter 5, a term-based representation has several deficiencies due to the fact that operations (e.g. methods in Java, procedures or functions in C, etc.) are typically not functions. Our techniques work over the space of operation sequences, a more expressive space than the space of terms.

One of the first papers to publish results on random testing for object-oriented programs was Doong and Frankl's ASTOOT [28]. ASTOOT is not a random testing tool, but a set of tools that generate test oracles for object-oriented methods based on user-provided algebraic specifications, execute tests, and report test execution results. In their experimental evaluation of the tools, Doong and Frankl used a random test input generator to create test inputs, and algebraic specifications to create oracles. They observed that longer tests were more effective than shorter tests in creating error-revealing tests inputs.

Recently, a number of techniques and tools have been developed for random testing object-oriented code. Csallner and Smaragdakis describe JCrasher, a tool that creates sequences of method calls for Java programs that the user can execute to discover exceptional behavior that can reveal errors [22]. JCrasher generates test inputs by building a "parameter graph" whose nodes are method signatures and edge between two nodes m_1 and m_2 mean that m_1 returns a value that can be used as input parameter to m_2. To create a test input, JCrasher selects a method to test, and uses the graph to find method calls whose return values can serve as input parameters to the method under test (test inputs that throw specific exceptions are labeled error-revealing). Reporting exceptions results in many false positives (in a run of JCrasher on a subset of the JDK, the vast majority of inputs tagged as error-revealing were false positives because they were illegal.)

We can view JCrasher as an instance of a top-down random test generator (Section 3.1), where the tool uses the parameter graph to find a path to the root of the method tree. To our knowledge, the distinction between top-down and bottom-up random test generation has not been drawn before, and an interesting open question is what are the merits of each approach. Bottom-up generation can be beneficial because it amenable to pruning techniques (like the ones presented in the thesis) that extend an existing sequence of operations based on the execution results of the sequence. The top-down approach is less amenable to this approach; on the other hand, top-down generation approach can be more focused when the goal is to create test inputs for a specific method under test.

JCrasher focus is on creating test inputs that throw exceptions and can report many false positives resulting from exceptions that do not constitute erroneous behavior. RANDOOP's approach is the opposite of JCrasher's in this respect: it *never* reports test inputs that throw exceptions, uses an extensible set of default contracts, and reports exceptions only if the user explicitly specifies the exception type as being error-revealing (the default set is very small compared to JCrasher). In our experience the vast majority of exception-throwing test inputs are illegal uses of the interface, not errors, and inspecting a voluminous number of test inputs by hand is infeasible (indeed, the authors of CnC did not inspect all the error-revealing inputs reported by their tool on the benchmarks used to the evaluate it, because of the large number).

In previous work, we implemented Eclat [79], a tool that generates test cases likely to expose errors by performing random generation augmented by automated pruning based on execution results. Eclat's pruning discards sequences that appear to be illegal, because they make the program behave differently than a set of correct training runs did. The previous work focused on automated classification of tests with an oracle created automatically using dynamic invariant detection [31]. RANDOOP [82] focuses on generating a set of behaviorally-diverse test inputs, including avoiding redundant objects, repetition to generate low-likelihood sequences, oracles based on API contracts that can be extended by the user, and regression oracles that capture the behavior of a program when run on the generated input. Eclat's performance is sensitive to the quality of the sample execution given as an input to the tool. Since RANDOOP does not require a sample execution, it is not sensitive to this parameter.

Jartege [77] and AutoTest [65] generate random method calls but require a user-provided formal specification (in JML or Eiffel) to determine if an input is error-revealing; the tools do not use the specification to guide generation; the search is undirected. To our knowledge, these tools have only been applied to small examples. Parasoft's Jtest [83] is a commercial tool that also uses specifications to determine if an input is error-revealing; its input generation algorithm is not published.

Another line of research uses evolutionary algorithms [90, 94, 38] that assign a fitness function to method sequences based on coverage collected during execution. The execution can be driven by randomly-generated inputs.

Adaptive random testing [18] (ART) is a random selection technique. Given a set of test inputs, the first input is selected at random (and removed from) the candidate. Each subsequent step selects the candidate input whose distance is greatest from the last-chosen input. When evaluated on a small set of subjects, it was found to lead more quickly to revealing the first error. More recently, Ciupa et al. defined a similarity metric for objects based on their representation, which enables adaptive random testing to be applied to object-oriented software, and created the ARTOO tool that uses their metric to select a subset of randomly-generate test inputs [19]. They found that ART sometimes led the tool to reveal an error faster than random testing alone, but sometimes caused the tool to miss creating an error-revealing test that would be revealed via random generation alone.

Up to now we have focused on random testing for finding errors. Another question is whether random testing actually improves "reliability" or "confidence." in the tested software. Frankl et al. provide a theoretical treatment of the subject in which they introduce the concept of "delivered reliability." [35], i.e. the likelihood of a program failing during actual use, and theoretically compare the delivered reliability of two approaches to testing: debug testing, which focuses on generating error-revealing test cases, and operational testing, which exercises the software using inputs randomly selected from an operation profile of expected use. Their theoretical results do not suggest one approach to be superior to the other.

11.3 Systematic Testing

This section surveys automated test generation techniques that do not use randomness as part of their generation strategy. We limit our discussion to techniques that generate test inputs for object-oriented units. We divide deterministic techniques into two broad categories. Techniques based on exhaustive exploration exhaustively explore the input space of the software under test, up to some bound. Techniques based on symbolic execution simulate the software's execution through a specific program path using symbolic values instead of concrete values, collect constraints over the symbolic values during the simulated execution, and attempt to solve the constraints to yield concrete inputs that follow the path.

11.3.1 Techniques Based on Exhaustive Exploration

Techniques based on exhaustive exploration consider all possible inputs up to a given size. For example, in the context of method sequence generation, an exhaustive exploration technique might execute all method sequences up to some given length, and execute them to determine if they violate the test oracle.

Visser et al. [92] use the Java PathFinder model checker (JPF [93]) to exhaustively explore method sequences. They consider sequences containing a single container object for integers, e.g. a list or a set. JPF takes as input a Java program and explores all possible executions of the program; a program may have more than one possible execution because of nondeterminism involving thread scheduling or random choices. To explore method sequences, they write a "driver program" for each class that (1) initializes the container, and (2) enters a loop that repeatedly and randomly selects a method to call, and an integer argument.

In their evaluation Visser et al. compare a number of state matching techniques that aim to avoid generating redundant inputs. Their results suggest that unsound state matching heuristics (heuristics that may cause the analysis to fail to explore distinct program behaviors) are effective in reducing the time required by the exhaustive exploration without affecting the code coverage achieved. These observations are similar to Xie et al.'s evaluation of several techniques that discard potentially redundant method sequences [96].

A different approach to test input generation for methods is to create input objects by directly setting their fields to specific values. The TestEra [61] and Korat [16] tools perform exhaustive exploration of Java data structures. TestEra [61] generates all non-isomorphic test inputs of a given size (e.g. all binary trees with 3 nodes), and checks a user-specified correctness property (e.g. that an representation invariant is preserved by the binary tree methods). The user of TestEra expresses the correctness criteria to check and a translation (an abstraction function) between the Java data structure and a model of the structure in Alloy, a first-order relational language [54]. An analyzer procedure for Alloy checks the model and reports any counterexamples to the correctness property. If no counterexamples are found, the analysis guarantees that the property is preserved (subject to the correctness of the translation).

Korat [16] is a successor to TestEra that generates all data structures up to some given size, without using a model finder like the Alloy analyzer. The user or Korat writes a "repOk" method for the data structure. The repOk method returns a boolean value, and represents a predicate that specifies the representation invariant of the data structure. Korat exhaustively generates all non-isomorphic data structures that satisfy the invariant. The technique uses the implementation of the repOk method to guide the generation, and avoids exploring structures that do not satisfy the invariant.

The two approaches discussed above—generating inputs by creating method sequences versus setting object fields—have advantages and disadvantages relative to each other. The former requires little or no manual work on the part of the user, while the latter may require the user to specify a translation function or a detailed representation invariant. However, putting an object in a particular state may require a long sequence of calls but a small number of explicit field assignments. We know of no published experimental comparisons of the two approaches.

Exhaustive test input exploration based on model checking is an active research field. Many of the techniques explored in context outside object-oriented software testing have not yet been translated to the object-oriented domain, including additional heuristics to avoid redundant inputs [13] or to guide the search in order to achieve greater coverage [43, 37].

11.3.2 Techniques Based on Symbolic Execution

Symbolic execution, first proposed by King [62], consists in executing the software under test using symbolic values instead of concrete values, for example `s.add(x)` instead of `s.add(3)`. Doing this requires a special symbolic execution engine that can execute using symbolic values. When the execution encounters a branch that depends on a symbolic value, the engine selects a branch direction and adds the corresponding constraint in terms of the symbolic value (e..g. "$x < 0$") to a set of path constraints. Solving the set of constraints resulting from the symbolic execution of a given program path results in concrete values that drive execution through the path.

A test generation technique based on symbolic-execution typically has a coverage goal in mind, such as branch or path coverage. It executes the program on symbolic input values, exploring program paths that satisfy the coverage goal (e.g. paths that covers all the branches). Finally, it uses a constraint solver to generate concrete inputs that drive execution through the paths explored by the symbolic execution engine. Symstra [97] uses symbolic execution to generate test inputs for Java container data structures (similar to those studied by Visser [92]). XRT [42] performs symbolic execution for .NET programs; Tillman et al. use XRT to generate tests using a user-provided unit test "template" called a parameterized unit test, which can contain concrete and symbolic values [89]. The tool attempts to generate fully concrete versions of the tests that achieve high code coverage. Check-n-Crash [23] uses ESC/Java to create abstract constraints over inputs that would lead to exceptional behavior, and uses a constraint solver to derive concrete test inputs that exhibit the behavior. DSD [24] augments Check-n-Crash with a dynamic analysis that it uses to filter out illegal input parameters, in a way similar to Eclat [79].

11.4 Comparisons of Random and Systematic Testing

There is a long-standing controversy in the testing research community regarding the relative merits of random testing and systematic testing, with both theoretical and empirical studies reaching conflicting conclusions. Some empirical studies [32, 4, 92, 17] suggest that random testing is not as effective (in terms of achieved coverage) as other test generation techniques, including chaining [32], exhaustive generation [4], model checking [92], and symbolic execution [92]. Ferguson and Korel compared the coverage achieved by inputs generated using their chaining technique versus randomly generated inputs [32]. Marinov et al. [4] compared the rate of mutants killed by a set of exhaustively generated test inputs with a random subset of the inputs. Visser et al. [92] compared the block and a form of predicate coverage [11] achieved by model checking, symbolic execution, and random test generation. In all three studies, random testing achieved less coverage than the systematic techniques.

It is difficult to generalize the results of these studies with regard to the relative advantages of random and systematic testing. The evaluations were performed on very small programs. Because the small programs apparently contained no errors, the comparison was in terms of coverage or rate of mutant killing [76], not in terms of true error detection, which is the best measure to evaluate test input generation techniques. While the systematic techniques used sophisticated heuristics to make them more effective, the random testing used for comparison is unguided random testing, with no heuristics to guide its search, or where random inputs are selected from among those generated by the systematic technique.

Theoretical studies suggest that random test generation is as effective as more systematic techniques such as partition testing [29, 30, 47, 75], and a growing number of empirical studies [47, 75, 44, 82] suggest that random testing's speed, scalability, and unbiased search make it an effective error-detection technique able to outperform systematic techniques. For example, in previous work we compared random and symbolic input generation and classification techniques on a set of programs. We found that random and systematic techniques are complementary: random test generation finds errors that symbolic execution cannot find because of limitations in symbolic execution engines or constraint solvers, and symbolic execution finds errors that random test generation cannot find because of highly specific constraints on the inputs [26]. We conjectured that random generation may benefit from using repetition; this was the motivation for implementing repetition in Randoop.

There are few case studies of random test generation's use in industry. Recent work by Groce et al. [44] describes a realistic application of random testing to a file system used in flight software. Their experience was similar to ours: random test generation found a large number of errors in a relatively short amount of time. Like Randoop, their tool eventually reached a plateau. To find more errors, Groce et al. suggest moving towards formal verification of the software; this is a different, and complementary approach to our attack on the plateau via techniques to increase fairness. They say that model checking would not scale to testing code for complex flight systems, while random testing scales and finds many errors in the code.

11.5 Hybrid Techniques

Dynamic test generation techniques combine concrete and symbolic input generation. Ferguson and Korel [32] describe “chaining,” a technique that executes the software on a concrete input and records the concrete program path that the execution follows. The technique then attempts to modify the input to cause the execution to follow a different branch for one of the conditions in the concrete path. To achieve this, the technique solves the symbolic constraints corresponding to the concrete path, but with one of the constraints negated (the one corresponding to the desired branch).

Godefroid et al.’s DART [39] and and Sen et al.’s concolic testing [86] are similar to chaining, but use a randomly-generated input as the initial concrete input used to initialize the search, and revert to concrete execution when the constraint solver fails or when the code cannot be handled by the symbolic execution engine (e.g. during native operating system calls). Majumdar et al. [69] describe hybrid concolic testing, which starts by performing undirected random test generation, and transitions to concolic testing only when random test generation fails to achieve new coverage. This approach is similar to our proposed branch-directed generation technique, but we propose to modify inputs via random educated guesses based on data flow analysis, not via symbolic execution.

Like directed random testing, our branch-directed generation technique (Chapter 6) is a *dynamic* generation technique, meaning that it uses runtime information about the software under test to guide the generation of new inputs. It is more closely related to techniques like DART [39] and concolic testing [86], which execute the software on random inputs, collect symbolic constraints at conditional statements, and use a constraint solver to modify the inputs in order to drive execution through uncovered branches. Instead of using a constraint solver to modify a test input so that it follows a desired branch, our technique tracks the flow and interactions of values during the execution of the sequence, and uses this information to determine influencing variables in the sequence, and to compute a set of values to set the influencing variables to.

Commercial tools like Jtest and Agitar’s Agitator [1] also perform combinations of concrete and symbolic execution; the details of their implementation are unpublished. Agitator creates test inputs using a variety of techniques, including random generation and data flow analysis, and proposes to the user program invariants based on execution of the test inputs.

Chapter 12

Conclusion

Directed random testing is automated, scalable, and it outputs compilable test cases that reveal errors in large, well-tested and widely-deployed applications. In an industrial environment, the technique led to a highly productive period of error discovery when used by a team of engineers testing a component that had already undergone 200 man years of testing; in addition, the technique revealed errors in the test team's existing testing and analysis tools, inadequacies in their manual testing, and even led them to improve their best practices. Since the study was conducted, Randoop has been added to an internal list recommended testing tools at Microsoft, and other test teams at the company have used Randoop to discover errors in software components and improve the effectiveness of their testing efforts. These results provide evidence of the promise of directed random testing in improving the productivity of test engineers and the quality of the software under test.

In the introduction, we described directed random testing as a technique that combines ideas from two approaches, random testing and systematic testing. Directed random testing adopts random testing's idea of randomly assembling inputs, and augments it with systematic testing's idea of pruning the input space to avoid generating useless inputs. Ours is not the only technique to combine elements of random and systematic testing; for example, recently-developed techniques like DART [39] and concolic testing [86] initialize a systematic generation technique with randomly-generated seed inputs. The key difference, and innovation, in directed random testing, is the point it occupies in the random-systematic spectrum of techniques. Other hybrid techniques are primarily systematic in their generation, and use randomness only to initialize the search. In contrast, our approach is primarily random in its generation, and incorporates systematicity only to guide the generation process. In exploring the "random end" of the random-systematic spectrum, we wanted to start with a simple and effective approach, random testing, and scaling it up to mitigate its deficiencies.

The exchange of ideas between the random and systematic approaches can benefit both communities, and there are many exciting research projects based on exploring different points in the spectrum. Groce et al. propose structural heuristics [43] to guide a model checker; the heuristics might also help a random test generator. Going the other way, our ideas about generation using a component set, or heuristic equivalence, could be translated into the exhaustive testing domain. Combining random and systematic approaches can result in techniques that retain the best of each approach. The exhaustive testing community can also benefit from research in random testing by learning when non-exhaustive techniques can yield comparable (or even better) results, as appears to be the case when the goal is achieving code coverage for some container data structures. This might point the way to the need for better benchmarks that differentiate the relative strengths of exhaustive and random testing.

Our vision for the future of software testing is one in which the generation of test inputs is largely automated by techniques that combine random, systematic, and other approaches to provide input sets of equal or better quality than those an engineer could produce by hand. Such a future would let test engineers focus effort on thinking about the properties that a software artifact should satisfy, providing powerful mechanized assistants that help them gain confidence that the properties are indeed satisfied by the software artifact under test. The work we have presented is a step toward this goal.

Bibliography

[1] Agitar web page. `http://www.agitar.com/`.

[2] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *PLDI '90: Proceedings of the ACM SIGPLAN 1990 conference on programming language design and implementation*, pages 246–256, New York, NY, USA, 1990. ACM.

[3] Paul Ammann and Jeff Offutt. *Introduction to Software testing.* Cambridge University Press, Cambridge, UK, February 2008.

[4] Alexandr Andoni, Dumitru Daniliuc, Sarfraz Khurshid, and Darko Marinov. Evaluating the "Small Scope Hypothesis". Technical Report MIT/LCS/TR-921, Massachusetts Institute of Technology Laboratory for Computer Science, September 2003.

[5] James H. Andrews, Felix C. H. Li, and Tim Menzies. Nighthawk: a two-level genetic-random unit test data generator. In *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 144–153, New York, NY, USA, 2007. ACM.

[6] J.H. Andrews, A. Groce, M. Weston, and Ru-Gang Xu. Random test run length and effectiveness. *23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 19–28, Sept. 2008.

[7] J.H. Andrews, A. Groce, M. Weston, and Ru-Gang Xu. Random test run length and effectiveness. *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, pages 19–28, Sept. 2008.

[8] 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 *1st Workshop on Model-Based Testing and Object-Oriented Systems (M-TOOS)*, Portland, OR, October 23, 2006.

[9] 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*, Atlanta, GA, USA, November 7–9, 2007.

[10] Alberto Bacchelli, Paolo Ciancarini, and Davide Rossi. On the effectiveness of manual and automatic unit test generation. In *ICSEA 2008: The Third International Conference on Software Engineering Advances*, 2008.

[11] Thomas Ball. A theory of predicate-complete test coverage and generation. In *Third International Symposium on Formal Methods for Components and Objects*, pages 1–22, The Netherlands, 2004.

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

[13] Dirk Beyer, Adam J. Chlipala, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. Generating tests from counterexamples. In *Proceedings of the 26th IEEE International Conference on Software Engineering (ICSE 2004, Edinburgh, May 26-28)*, pages 326–335. IEEE Computer Society Press, Los Alamitos (CA), 2004.

[14] David L Bird and Carlos Urias Munoz. Automatic generation of random self-checking test cases. *IBM Systems Journal*, 22(3):229–245, 1983.

[15] Joshua Bloch. *Effective Java (2nd Edition) (The Java Series).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2008.

[16] Chandrasekhar Boyapati, Sarfraz Khurshid, and Darko Marinov. Korat: automated testing based on Java predicates. In *ISSTA '02: Proceedings of the 2002 ACM SIGSOFT international symposium on Software testing and analysis*, pages 123–133, New York, NY, USA, 2002. ACM Press.

[17] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. Exe: automatically generating inputs of death. In *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, pages 322–335, New York, NY, USA, 2006. ACM Press.

[18] Kwok Ping Chan, Tsong Yueh Chen, Fei-Ching Kuo, and Dave Towey. A revisit of adaptive random testing by restriction. In *COMPSAC '04: Proceedings of the 28th Annual International Computer Software and Applications Conference (COMPSAC'04)*, pages 78–85, Washington, DC, USA, 2004. IEEE Computer Society.

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

[20] Koen Claessen and John Hughes. Quickcheck: a lightweight tool for random testing of haskell programs. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 268–279, New York, NY, USA, 2000. ACM.

[21] E. M. Clarke, Orna Grumberg, and Doron Peled. *Model Checking.* MIT Press, 1999.

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

[23] 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, New York, NY, USA, 2005. ACM.

[24] Christoph Csallner, Yannis Smaragdakis, and Tao Xie. DSD-Crasher: A hybrid analysis tool for bug finding. *ACM Trans. Software Engineering Methodology*, 17(2):1–37, 2008.

[25] Marcelo d'Amorim, Cristiano Bertolini, and Juliano Iyoda. Change-based random sequence generation for efficient state-space exploration of object-oriented programs. Federal University of Pernambuco. Personal communication, 2009.

[26] Marcelo d'Amorim, Carlos Pacheco, Darko Marinov, Tao Xie, and Michael D. Ernst. An empirical comparison of automated generation and classification techniques for object-oriented unit testing. In *Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006)*, September 2006.

[27] Brett Daniel and Marat Boshernitsan. Predicting effectiveness of automatic testing tools. In *ASE 2008: 23rd IEEE/ACM International Conference on Automated Software Engineering*, New York, NY, USA, sep 2008. ACM.

[28] Roong-Ko Doong and Phyllis G. Frankl. The ASTOOT approach to testing object-oriented programs. *ACM Transactions on Software Engineering Methodology*, 3(2):101–130, 1994.

[29] Joe W. Duran and Simeon Ntafos. A report on random testing. In *Proceedings of the 5th International Conference on Software Engineering*, pages 179–183. IEEE Press, 1981.

[30] Joe W. Duran and Simeon C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–444, July 1984.

[31] Michael D. Ernst, Jeff H. Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, and Chen Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 69(1–3):35–45, December 2007.

[32] Roger Ferguson and Bogdan Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering and Methodology*, 5(1):63–86, January 1996.

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

[34] Justin E. Forrester and Barton P. Miller. An empirical study of the robustness of Windows NT applications using random testing. In *4th USENIX Windows System Symposium*, pages 59–68, Seattle, WA, USA, August 2000.

[35] Phyllis G. Frankl, Richard G. Hamlet, Bev Littlewood, and Lorenzo Strigini. Evaluating testing methods by delivered reliability. *IEEE Transactions on Software Engineering*, 24(8):586–601, August 1998.

[36] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley professional computing series, 1994.

[37] Patrice Godefroid and Sarfraz Khurshid. Exploring very large state spaces using genetic algorithms. In *TACAS '02: Proceedings of the 8th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 266–280, London, UK, 2002. Springer-Verlag.

[38] Patrice Godefroid and Sarfraz Khurshid. Exploring very large state spaces using genetic algorithms. *International Journal on Software Tools for Technology Transfer*, 6(2):117–127, 2004.

[39] Patrice Godefroid, Nils Klarlund, and Koushik Sen. Dart: directed automated random testing. In *PLDI '05: Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[40] James Gosling, Ken Arnold, and David Holmes. *The Java Programming Language, 4th Edition*. Prentice Hall PTR, August 2005.

[41] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *Java Language Specification, Second Edition: The Java Series*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[42] Wolfgang Grieskamp, Nikolai Tillmann, and Wolfram Schulte. XRT - Exploring Runtime for .NET - Architecture and Applications. In *SoftMC 2005: Workshop on Software Model Checking*, Electronic Notes in Theoretical Computer Science, July 2005.

[43] Alex Groce and Willem Visser. Model checking Java programs using structural heuristics. *SIGSOFT Software Engineering Notes*, 27(4):12–21, 2002.

[44] Alex D. Groce, Gerard Holzmann, and Rajeev Joshi. Randomized differential testing as a prelude to formal verification. In *ICSE '07: Proceedings of the 29th International Conference on Software Engineering*, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[45] Philip J. Guo, Jeff H. Perkins, Stephen McCamant, and Michael D. Ernst. Dynamic inference of abstract types. In *ISSTA 2006, Proceedings of the 2006 International Symposium on Software Testing and Analysis*, pages 255–265, Portland, ME, USA, July 18–20, 2006.

[46] Dick Hamlet. Random testing. In *Encyclopedia of Software Engineering*. John Wiley and Sons, 1994.

[47] Dick Hamlet and Ross Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, December 1990.

[48] K. V. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):242–257, 1970.

[49] A. Hartman. Personal communication with Michael D. Ernst, July 2006.

[50] Johannes Henkel and Amer Diwan. Discovering algebraic specifications from Java classes. In Luca Cardelli, editor, *ECOOP 2003 - Object-Oriented Programming, 17th European Conference*, Darmstadt, July 2003. Springer.

[51] John L. Henning. Spec cpu2000: Measuring CPU performance in the new millennium. *Computer*, 33(7):28–35, 2000.

[52] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std. 610.12-1990*, 10 Dec 1990.

[53] Kobi Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proc. 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE 2008)*, September 2008.

[54] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2006.

[55] Daniel Jackson, Martyn Thomas, and Lynette I. Millett (Editors). *Software for Dependable Systems: Sufficient Evidence?* National Academies, Washington, DC, May 2007.

[56] Java Compatibility Kit web page. `https://jck.dev.java.net/`.

[57] Hojun Jaygari and Kim Sung. Coverage based selection in random testing for Java. Iowa State University. Personal communication, 2009.

[58] Hojun Jaygari and Kim Sung. Test case reduction in adaptive random testing for Java. Iowa State University. Personal communication, 2009.

[59] Cem Kaner. What is a good test case? In *Software Testing Analysis and Review Conference (STAR) East*, Orlando, FL, May 2003.

[60] Sarfraz Khurshid. *Generating structurally complex tests from declarative constraints.* PhD thesis, Massachusetts Institute of Technology, December 2003. Supervisor: Daniel Jackson.

[61] Sarfraz Khurshid and Darko Marinov. TestEra: Specification-based testing of Java programs using SAT. *Automated Software Engineering*, 11(4):403–434, 2004.

[62] James C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[63] N. P. Kropp, P. J. Koopman, and D. P. Siewiorek. Automated robustness testing of off-the-shelf software components. In *FTCS '98: Proceedings of the The Twenty-Eighth Annual International Symposium on Fault-Tolerant Computing*, page 230, Washington, DC, USA, 1998. IEEE Computer Society.

[64] Shuvendu Lahiri. Personal communication, January 2009.

[65] A. Leitner, I. Ciupa, B. Meyer, and M. Howard. Reconciling manual and automated testing: the autotest experience. In *Proceedings of the 40th Hawaii International Conference on System Sciences - 2007, Software Technology*, January 3-6 2007.

[66] Barbara Liskov and John Guttag. *Program Development in Java: Abstraction, Specification, and Object-Oriented Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[67] Jacques Loeckx, Hans-Dieter Ehrich, and Markus Wolf. *Specification of abstract data types*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[68] Panagiotis Louridas. Junit: Unit testing and coding in tandem. *IEEE Software*, 22(4):12–15, 2005.

[69] Rupak Majumdar and Koushik Sen. Hybrid concolic testing. In *ICSE '07: Proceedings of the 29th international conference on Software Engineering*, pages 416–426, Washington, DC, USA, 2007. IEEE Computer Society.

[70] Microsoft. Test Release Criteria for Small IT Solution product. TechNet note (`http://technet.microsoft.com/`), July 2004.

[71] Barton P. Miller, Gregory Cooksey, and Fredrick Moore. An empirical study of the robustness of macos applications using random testing. In *RT '06: Proceedings of the 1st international workshop on Random testing*, pages 46–54, New York, NY, USA, 2006. ACM Press.

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

[73] Glenford J. Myers and Corey Sandler. *The Art of Software Testing*. John Wiley & Sons, 2004.

[74] .NET Framework web page. `http://www.microsoft.com/net/`.

[75] Simeon Ntafos. On random and partition testing. In *ISSTA '98: Proceedings of the 1998 ACM SIGSOFT international symposium on Software testing and analysis*, pages 42–48, New York, NY, USA, 1998. ACM Press.

[76] J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In *Mutation 2000: Mutation Testing in the Twentieth and the Twenty First Centuries*, pages 45–55, October 2000.

[77] Catherine Oriat. Jartege: A tool for random generation of unit tests for Java classes. In *Quality of Sofware Architectures and Software Quality, 2nd International Workshop of Software Quality - SOQUA'05, Lecture Notes in Computer Science 3712*, pages 242–256, September 2005.

[78] Alessandro Orso and Tao Xie. Bert: Behavioral regression testing. In *WODA '08: Proceedings of the 2008 international workshop on dynamic analysis*, pages 36–42, New York, NY, USA, 2008. ACM.

[79] 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.

[80] Carlos Pacheco, Shuvendu Lahiri, Michael Ernst, and Thomas Ball. Feedback-directed random test generation. Technical Report MSR-TR-2006-125, Microsoft Research, September 2006.

[81] Carlos Pacheco, Shuvendu K. Lahiri, and Thomas Ball. Finding errors in .NET with feedback-directed random testing. In *ISSTA 2008: International Symposium on Software Testing and Analysis*, Seattle, Washington, July 20–24, 2008. To appear.

[82] 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*, Minneapolis, MN, USA, 2007. IEEE Computer Society.

[83] Parasoft web page. `http://www.parasoft.com/`.

[84] Randoop web page. `http://people.csail.mit.edu/cpacheco/randoop/`.

[85] Derek Rayside, Zev Benjamin, Rishabh Singh, Joseph Near, Aleksandar Milicevic, and Daniel Jackson. Equality and hashing for (almost) free: Generating implementations from abstraction functions. In *ICSE '09: Proceedings of the 31st international conference on software engineering*. ACM, 2009.

[86] Koushik Sen and Gul Agha. CUTE and jCUTE : Concolic unit testing and explicit path model-checking tools. In *18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 419–423. Springer, 2006. (Tool Paper).

[87] Kevin Sullivan, Jinlin Yang, David Coppit, Sarfraz Khurshid, and Daniel Jackson. Software assurance by bounded exhaustive testing. *SIGSOFT Software Engineering Notes*, 29(4):133–142, 2004.

[88] Sun's Bug Database web page. `http://bugs.sun.com/`.

[89] Nikolai Tillmann and Wolfram Schulte. Unit tests reloaded: Parameterized unit testing with symbolic execution. *IEEE Software*, 23(4):38–47, 2006.

[90] Paolo Tonella. Evolutionary testing of classes. In *ISSTA '04: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis*, pages 119–128, New York, NY, USA, 2004. ACM Press.

[91] Willem Visser, Klaus Havelund, Guillaume Brat, Seungjoon Park, and Flavio Lerda. Model checking programs. *Automated Software Engineering*, 10(2):203–232, 2003.

[92] Willem Visser, Corina S. Păsăreanu, and Radek Pelánek. Test input generation for Java containers using state matching. In *ISSTA'06: Proceedings of the 2006 international symposium on Software testing and analysis*, pages 37–48, New York, NY, USA, 2006. ACM Press.

[93] Willem Visser, Corina S. Păsăreanu, and Sarfraz Khurshid. Test input generation with Java Pathfinder. *SIGSOFT Software Engineering Notes*, 29(4):97–107, 2004.

[94] Stefan Wappler and Frank Lammermann. Using evolutionary algorithms for the unit testing of object-oriented software. In *GECCO '05: Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 1053–1060, 2005.

[95] E.J. Weyuker and T.J. Ostrand. Theories of program testing and the application of revealing subdomains. *Software Engineering, IEEE Transactions on*, SE-6(3):236–246, May 1980.

[96] Tao Xie, Darko Marinov, and David Notkin. Rostra: A framework for detecting redundant object-oriented unit tests. In *ASE '04: Proceedings of the 19th IEEE international conference on Automated software engineering*, pages 196–205, Washington, DC, USA, 2004. IEEE Computer Society.

[97] Tao Xie, Darko Marinov, Wolfram Schulte, and David Notkin. Symstra: A framework for generating object-oriented unit tests using symbolic execution. In *TACAS*, pages 365–381, April 2005.

[98] Yuan Yu, Tom Rodeheffer, and Wei Chen. Racetrack: efficient detection of data race conditions via adaptive tracking. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 221–234, New York, NY, USA, 2005. ACM.

[99] Andreas Zeller. Yesterday, my program worked. today, it does not. why? *SIGSOFT Software Engineering Notes*, 24(6):253–267, 1999.

Appendix A

Code Listing for Polynomial Classes

```
public class Rat {

  public int num;
  public int den;
  public double approx;

  // Creates a Rat representing n/d. Requires: d is not 0.
  public Rat(int n, int d) {
    if (d == 0) {
      num = 0; den = 1;
      approx = this.approx();
      return;
    }
    // reduce ratio to lowest terms
    int a = n, b = d;
    while (b != 0) {
      int t = b; b = a % b; a = t;
    }
    num = n / a; den = d / a;
    if (den < 0) {
      num = -num; den = -den;
    }
    approx = this.approx();
  }

  // Approximates the value of this rational.
  public double approx() {
    return ((double)num) / ((double)den);
  }

  // Returns the addition of this and arg.
  public Rat add(Rat arg) {
    // a/b + x/y = (ay + bx)/by
    return new Rat(this.num*arg.den + arg.num*this.den, this.den*arg.den);
  }

  // Return a new rational equal to "this-arg".
  public Rat sub(Rat arg) {
    return add(new Rat(-arg.num, arg.den));
  }

  // Returns true if this represents the rational "0".
  public boolean isZero() {
    return num == 0;
  }

  public boolean equals(Object o) {
    if (o == null) return false;
    if (o == this) return true;
    Rat r = (Rat)o;
    return r.num == this.num && r.den == this.den;
  }

  public int hashCode() {
    int hash = 7;
    hash = hash * 31 + num;
    hash = hash * 31 + den;
    return hash;
  }
}

public class Mono {

  // coefficient
  public final Rat coeff;
  // exponent
  public final int exp;

  // Creates a monomial with the specified coefficient
  // and exponent. Requires: c is not null, e is not negative.
  public Mono(Rat coeff, int exp) {
    if (coeff == null || exp < 0)
      throw new IllegalArgumentException("Invalid arguments.");
    this.coeff = coeff;
    this.exp = exp;
  }

  public boolean equals(Object o) {
    if (this == o) return true;
    if (o == null  || !(o instanceof Mono)) return false;
    Mono m = (Mono)o;
    return this.coeff.equals(m.coeff) && this.exp == m.exp;
  }

  public int hashCode() {
    int hash = 7;
    hash = hash * 31 + coeff.hashCode();
    hash = hash * 31 + exp;
    return hash;
  }
}

public class Poly {
  public Mono[] ms;

  // Creates a new polynomial with the specified elements.
  // Requires: elements array is not null, elements are sorted
  // in order of decreasing exponent, and have non-zero coefficient.
  public Poly(Mono... monos) {
    ms = new Mono[monos.length];
    for (int i=0;i<ms.length;i++)
      ms[i] = monos[i];
    assert this.repOk();
  }

  // Adds m to this polynomial, and
  // returns the result. Requires:
  // m is not null.
  public Poly add(Mono m) {

    // Will contain new monomial elements.
    List<Mono> mList=new ArrayList<Mono>();

    // Go through the elements of this
    // polynomial, inserting the monomial m
    // in the appropriate place in mList.
    for (int i=0;i<ms.length;i++) {
      if (ms[i].exp == m.exp) {
        Rat sum = ms[i].coeff.add(m.coeff);
        Mono msum = new Mono(sum, m.exp);
        mList.add(msum); // ERROR
        continue;
      }
      if (ms[i].exp < m.exp) {
        mList.add(m);
      }
      mList.add(ms[i]);
    }
    Poly addition = new Poly(mList.toArray(new Mono[0]));
    assert addition.repOk();
    return addition;
  }

  // Checks representation invariant:
  // 1. Elements sorted in order of decreasing exponent.
  // 2. No two monomials have the same exponent.
  // 3. No monomial has zero coefficient.
  boolean repOk() {
    for (int i=0 ; i<ms.length ; i++) {
      if (i > 0 && ms[i-1].exp <= ms[i].exp)
        return false;
      if (ms[i].coeff.num == 0)
        return false;
    }
    return true;
  }
}
```